

MarkUp

Data structures

Particle effects

GMPhysics

Introduction to DLL making

Level editors

Smooth online movement

And Much More!



Interviews



Resources



Previews



Reviews



Tutorials

Future-proofing

How can you ensure your creation will be around 10 years down the road and **still** be enjoyed? That's the topic for this month's *Editor's Desk*.

This month was a bit different, as you probably guessed by the late release date, mainly I headed off on a two month vacation and passed the mag onto Eyas, who then left for vacation and passed it onto Andris, who then left. Long story short I'm finishing the mag on my vacation time ☺. We should be back on track for next issue, just keep those articles coming in!

But, heading back to the original question, how can you "future-proof" your game and have it available for years to come? It's a tough job, and it has to start way back in the planning stage with you subject matter and target audience. Does your audience need knowledge of a current event (e.g. a sporting event?). Is the subject matter time-specific? This will limit the enjoyable life of your creation.

There is also the development side;

your choice of programming language and development environment will affect the about of time the game will remain operable. Game Maker has a good record here, with a game made in GM v6 being playable on Windows 98 to Windows Vista, a time frame of 10 years.

But, what can you do if you make a game, and there isn't a great interest in it, or you are no longer able to maintain it? How can you keep the game progressing? Open Source it! Release it under a license like the GPL or Creative Commons and let other continue to learn from it and improve upon it. Your work lives on!

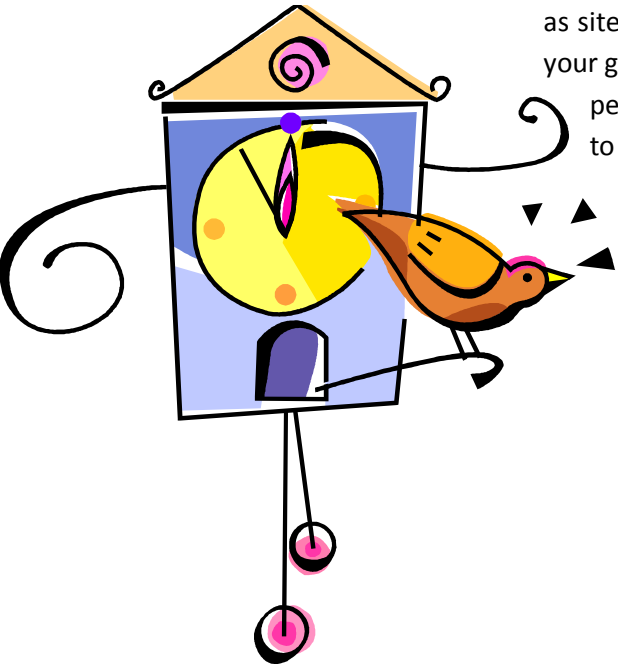
Not to mention open sourcing your work under one of the aforementioned licenses from the get go will allow you to get contributions and patches from fellow game developers, and can also open up the opportunity for you to use code and resources from other open source games in yours.

Another way to keep your game living on is to have it hosted on many sites, so as sites begin to die out you'll still have your game hosted elsewhere, and people will still have the opportunity to download it.

Keep up the games! I'll see you next month (hopefully ☺).



Robin Monks,
Editor



Contributors This Issue

Robin Monks	Editor
Eyas Sharaiha	Editor
Andris Belinskis	Editor
Leif Greenman	Writer
Rhys Andrews	Writer
José Méndez	Writer
Philip Gamble	Writer
Gregory	Writer
Bart Teunis	Writer
Sean Flanagan	Writer
Jake	Writer
Stephan	Writer
Dan Meinzer	Writer & Graphic Designer

Table of Contents

Editorials	
Future-proofing	2
Clone Games.....	6
Make your code easier to read	10
Promoting your game to the GMC..	20
Tutorials	
Particle Effects Tutorial	3
Real time blur	7
Cartesian and Polar Coordinates	9
Binary for Beginners.....	11
Coding Styles	14
Modulo in Computer Science	15
Smooth Online Movement.....	17
Introduction to Making DLLs	19
Level Editors	22
GMPhysics	25
Graphic Design.....	31
Sound Effects: How Far?	32
Reviews	
Textbar Maker	14
FATAL.....	33
Bounce 2.....	34
Snow Ball War.....	35
Monthly Specials	
Extension of the Month.....	29
Script of the Month.....	30

MarkUp is a gmking.org publication; please visit GMking for more free game development resources!

Photo © 2007 Robin Monks

Introduction

Particle Effects are one of the most well known visual/special effect methods used in GML, and in many other languages as well. Particles are various shapes (and sizes), with very little information. This information includes where they're supposed to go, their appearance, and more. With such little information, particles take little CPU usage to calculate where and how to draw them on the screen. This is why particles are a great method to use if you want effects such as rain, explosions, smoke, fireworks, flames, and more. In the *GameCave Effects Engine*, many engines are created on particles alone (and sometimes with some non-particle effects attached). Unfortunately, Particle Systems/effects can take some time to get the hang of. They also take a lot of trial and error, and patience to get individual particle types the way you want them.

Inside a particle system are two types of elements. *Particle Types*, which are the particles themselves – what they look like, how they move, how they live/die, etc. Particle types are actually not a part of any system, and can be used by any system – however, for the sake of simplicity I like to think of particle types as part of a system. The other type is *filters*. Filters manipulate and change the life of the particle as it is displayed on the screen. The most commonly used filter is an emitter. Emitters create the particles in certain regions of a room. Other filters include attractors (which draw particles to a certain position with an amount of force), destroyers (which destroys particles that come

within a region in the room), deflectors (which bounces particles off an invisible force), and changers (which change particles into other types of particles when they collide with an invisible region/force). In this tutorial, only the emitter filter will be explained in detail.

Creating systems, filters, and types

A game can have as many systems, filters, and particle types as you like. Of course, the more you have, the more memory they take up. To create one of these, all you must do is use the following functions:

```
system0 = part_system_create();  
  
//Create a System  
emitter =  
part_emitter_create(system0);  
  
//Create an emitter in the system  
'system0'  
attractor =  
part_attractor_create(system0);
```

```
//Create an attractor in the system  
'system0'  
destroyer =  
part_destroyer_create(system0);  
//Create a destroyer in the system  
'system0'  
changer =  
part_changer_create(system0);  
//Create a changer in the system  
'system0'  
particle0 = part_type_create();  
//Create a particle type.
```

If you do not assign these functions to a variable, the system is still created however you will not be able to access it because the unique ID of the system, filter, or type was not saved to anything you can refer to. Remember, if you create a second system and assign it to the same 'system0' variable, the old system will technically be lost and you will not be able to do anything with the system.



Particle Effects Tutorial

As you can tell, systems and particles do not have any arguments. This is because they are not assigned to anything. Systems cover all the filters, and so filters must assign to the system but not vice versa. Particles are not part of any system, but filters inside systems (and systems themselves) affect them.

After you create systems, filters, and particle types, it's a good idea to start customizing them. This allows you to make the filters, particles, and systems behave as you want them to. In this tutorial, I will show some key functions and how to use them – however, to explore the boundaries just search for the function prefixes (i.e **part_type_** or **part_system_**) in the GM manual and you will find an index of all the functions.

Common System Functions

```
part_system_position(system0,x,y);
```

This presents the position of the system. All filters in the system that are positioned in the room somewhere are positioned relative to this functions x and y values. For instance, if the position of an emitter is from 5-10 (x) and 5-10 (y), and the position of the system is changed from its default 0,0 values to 5,5 - then the emitter will be placed at 10-15 (x) and 10-15 (y) on the room.

```
part_system_depth(system0,depth);
```

When particles are created from emitters inside a system or a system itself, this function will determine what depth the

particles have.

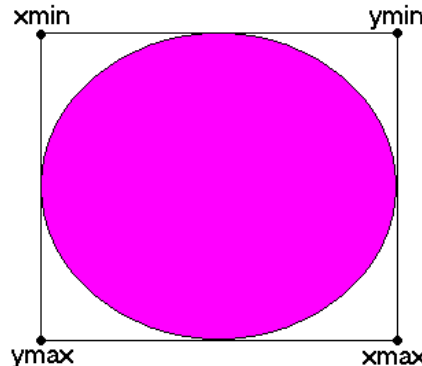
```
part_system_update(system0);
```

This handy function allows you to fast-forward the movement of particles. If you, for instance, create a snow effect, and the snowflakes are created around the outside of the room, you want to have the room start with snowflakes all over the room, instead of hanging around the outside of the room and slowly drifting in.

To do this, you use this function to move the particles 1 step forward (and of course repeat the function multiple times).

Common Emitter Functions

Assuming the Shape is **ps_shape_ellipse**



■ The emitters final region.

```
part_emitter_region(system0,
emitter, x, x, y, y, ps_shape_line,
ps_distr_linear);
```

This function is very important with any emitter. It determines the position of the emitter on the room (relative to the systems position) and how the particles are created from this emitter. The x and y arguments

determine the region where particles can be created. The x minimum and y minimum are like the top and left sides, while x maximum and y maximum are like the bottom and right sides. When the emitter releases particles, they can only be released within these sides. The last two arguments determine the shape and way of distributing particles within the x/y coordinates. A linear distribution

(**ps_distr_linear**) means the particles have equal chance of being created in all areas of the region, while a Gaussian distribution

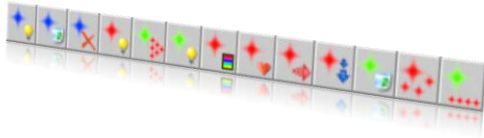
(**ps_distr_gaussian**) means that particles have more chance of being created right in the middle, and quadratic ally decrease chances as it gets to the edges of the region. See the diagram on the right to see how shapes fit with regions. Finally, an inverted Gaussian (**ps_distr_invgaussian**) has more chance of particles being created on the edges of the region than the centre.

Below is a list of possible shape and distribution constants that can be used in the last 2 arguments of this function

- **ps_shape_diamond**
- **ps_shape_ellipse**
- **ps_shape_line**
- **ps_shape_rectangle**
- **ps_distr_linear**
- **ps_distr_invgaussian**
- **ps_distr_gaussian**



Particle Effects Tutorial



```
part_emitter_burst(system0,emitter,p
article,10);

part_emitter_stream(system0,emitter,
particle,1);
```

These functions actually create certain particles into the emitter, using the region and any other specifications/customizations the emitter has had. The first function "bursts" a set amount of particles into the region. This doesn't affect how the particles move - it simply means one group of particles is created and then no more are created unless the function is called again. The last argument asks for how many particles to burst, while the second last argument asks for the ID of the particle you'd like to burst (the variable name that accompanied the creation function).

The STREAM function works very similarly, however stream continually creates more and more particles. Every step, in fact, a new batch of a specified amount of particles is created. If you want the streaming to go slowly, you can specify a negative number for the last argument; this means that 1 particle will be created on an average of steps. -5 will mean an average of 1 particle will be created every 5 steps.

Common Particle Type Functions

```
part_type_alpha1(particle0,1);

part_type_alpha2(particle0,1,0);

part_type_alpha3(particle0,1,0.5,0);
```

These 3 functions control the alpha (transparency) values of the particles. The first function gives a fixed alpha value that the particle withholds from its birth to its death. The second function uses a 'fade to' technique, in which the particle is created with the first alpha value and fades into the second, reaching the second as soon as it dies. The third function is similar to the second however it has a second key frame for the peak of its life.

```
part_type_blend(particle0,true);
```

This function can give the particle additive blending (the colors being drawn behind it are added to the color of the particle), or can take away the blending if given a false value.

```
part_type_color1(particle0, c_red);

part_type_color2(particle0, c_red,
c_white);

part_type_color3(particle0, c_red,
c_white, c_yellow);

part_type_color_mix(particle0,
c_red, c_white);
```

These four functions can determine the color of your particle. There are other functions but these are the most common. The first 3

functions work just like the 3 alpha functions, in that they give a fixed color for the birth of the particle, and the colour fades to the second and third key colours throughout its life. The fourth function gives 2 colors in which the particle must pick a colour between the two colours. For instance, with **c_red** and **c_white** as the mix, the particle can be anything between pure red and pure white, such as light red.

```
part_type_direction(particle0,0,360,
0,0);
```

This function gives the particle direction in its movement. A minimum and maximum direction can be given, plus an 'increment' value (how much the direction is increased by each step), and a 'wiggle' value (how much the direction sways from its midpoint).

```
part_type_shape(particle0,pt_shape_e
xplosion);
```

This function allows you to pick from the predefined shapes that GM supplies, to apply it to your particle. Without this the particle is a lousy pixel (or you can pick the particle as a lousy pixel using **pt_shape_pixel**). You can of course make your own shapes and use the **part_type_sprite** function instead. The following shape constants can be used:

- **pt_shape_circle**
- **pt_shape_cloud**
- **pt_shape_disk**



Particle Effects Tutorial

- `pt_shape_explosion`
- `pt_shape_flare`
- `pt_shape_line`
- `pt_shape_pixel`
- `pt_shape_ring`
- `pt_shape_smoke`
- `pt_shape_spark`
- `pt_shape_sphere`
- `pt_shape_square`
- `pt_shape_star`

```
part_type_size(particle0,1,2,0.1,0);
```

This function provides the particle with sizes. A minimum size, maximum size (the particle will pick an initial size between these two), a size increment (how much the size increases or decreases if a negative value is given), and a size 'wiggle'.

```
part_type_speed(particle0,5,10,1,0.5);
```

The function defines how fast the particle will go. Provided is a minimum and maximum speed for the particle to start with, a speed increment per step, and a

speed 'wiggle'.

```
part_type_life(particle0,50,100);
```

This function tells the particle its life span. You can give it a minimum life span, and a maximum life span - both in steps.

Global Common Functions

These functions are used in all filters, types, and systems (unless stated otherwise). They all perform similar "maintenance" actions. Replace (---) with the prefix, such as 'system', 'type', or 'emitter'.

```
part_---_destroy(system0);
```

Destroys (removes from the memory) the system, filter, or type.

```
part_---_clear(system0);
```

Reverts all the customizations of the filter, type, or system to its default settings.

```
part_---_exists(system0);
```

Returns whether the filter, type, or system exists.

Conclusion

I hope this gives you a bit of a better understanding in particles. It's quite hard to compress all the information about particle systems into 5 pages, and so I've only explained the most important parts. If you're at all interested, feel free to read my full, 17-page tutorial from <http://gmc.yoyogames.com/index.php?showtopic=272034>, or if you want to see some particle effects in action, you can download the GameCave Effects Engine from <http://gmc.yoyogames.com/index.php?showtopic=138220>

And remember, particles are very useful but use them in consideration – just like you would for laxatives.

Rhys Andrews■

Clone Games

EDITORIALS

"Clone games" is a subject that personally confuses me. On one hand it is true what they say: people who do clone games are just copying ideas of other games – they're not innovating.

But that's only one side of the story. It is true that people who make such games are not innovating – but that's only when it comes to game design.

There's so much more in a game than game design, this includes graphics,

sound effects, programming, and more.

There's also something else: and that is that making clone games requires a skill that no other type of games require: copying.

To make a good clone game, the elements of the game need to be replicated properly – the better the replication, the better the game.

Replication varies from using the same effects and timing, to more complex

tasks, such as creating a flawless battle engine, etc.

Basically, making clone games also requires talent – just a different type of talent.

I made it clear that there's nothing wrong with clone games, or people who make them – but does an average gamer want to play a clone game: usually, no.

Eyas Sharaiha■

Real-time Blur

Blur effects can make your game look quite professional if used well. There are several ways to achieve real-time blur in GM, here we are going to implement two: The first method is easy, cheap and quick, and the second one is slower and a bit more elaborated but will yield excellent results.

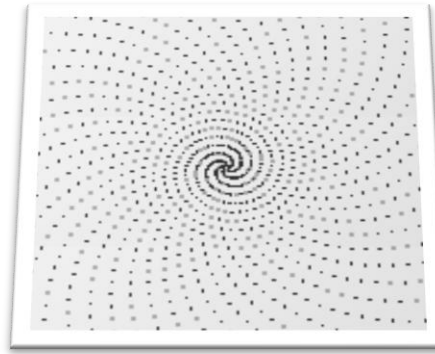
First Method: Jittering

Jittering consist in taking an image and draw it several times, in the same place, but moving it a bit each time in a different direction. We will draw the image with a very low alpha value, so that in the end, all the "stacked" images will look like a blurred version of the original.

So, we need a function to know where to draw the image each time. It should be as quick as possible to compute, you can use any kind of distribution, even random points. However some functions will make the blur look better than others. You can use this one, for example:

```
angle = 0;
spiral_spread = 1;
for(i = 0; i < 360 * iterations; i += 1) {
    posx = x + cos(angle) * spiral_spread;
    posy = y + sin(angle) * spiral_spread;
    angle += 5;
    spiral_spread += 0.1;
    draw_sprite_ext(sprite_index,
        image_index, blurx, blury,
        image_xscale, image_yscale,
        image_angle, c_white,
        image_alpha / (iterations * 0.3));
}
```

This one will draw the images following this spiral pattern: (each point represents the x,y coordinates at which a image will be drawn.)



Just store in an array a few of the drawing coordinates we produced using our function, so that we don't have to recompute them each time:

```
//initialize jitter array:
global.jitter1[0,0] = -0.334818; //x
coordinate 1
global.jitter1[0,1] = 0.435331; //y
coordinate 1
global.jitter1[1,0] = 0.286438; //x
coordinate 2
global.jitter1[1,1] = -0.393495; //y
coordinate 2
global.jitter1[2,0] = 0.459462;
global.jitter1[2,1] = 0.141540;
global.jitter1[3,0] = -0.414498;
global.jitter1[3,1] = -0.192829;
global.jitter1[4,0] = -0.183790;
global.jitter1[4,1] = 0.082102;
global.jitter1[5,0] = -0.079263;
global.jitter1[5,1] = -0.317383;
global.jitter1[6,0] = 0.102254;
global.jitter1[6,1] = 0.299133;
global.jitter1[7,0] = 0.164216;
global.jitter1[7,1] = -0.054399;
```

Next, we only need to draw the image several times (for this we will use a "for" statement) slightly altering the drawing coordinate using our precomputed values:

```
blur = 5;
for(i = 0; i < iterations; i += 1) {
    blurx = x + global.jitter1[i,0] * blur;
    blury = y + global.jitter1[i,1] * blur;
    draw_sprite_ext(sprite_index,
        image_index, blurx, blury,
        image_xscale, image_yscale,
        image_angle, c_white,
```

```
image_alpha / (iterations * 0.3));
}
```

That's it. Notice how I multiplied the precomputed values by a "blur" variable. That will allow us to control the amount of jitter applied (That is, the separation between drawing points).

Now, use the array initialization code at the beginning of the game (you can write the code in a script if you like, or use it as the room creation code of the first room), and put the drawing code in the draw event of the object you want to blur.

Execute it: As you can see, the quality is good for low blur values, but when we try to crank up the blur, it begins to look really ugly. So it is good only if you want a quick and subtle blur effect.

Second Method: Repeated Linear Filtering

I found this clever idea in an OpenGL tutorial, and I thought it could be applied to Game Maker. It takes advantage of a very common feature in modern graphic cards: linear texture filtering.

The trick is as follows: What happens if we take a 32x32 image (for example) and we divide its size by 2? We obtain a 16x16 image, with less quality than the original. Now, try to scale it again to 32x32. If we have linear filtering enabled, the computer will try to "soften" the resulting image to make the loss of quality produced when scaling less apparent.

That's the key: this automatic filtering process is really quick. So we could



Real-time Blur

repeat this process several times (scale the image down, scale the image up (this blurs the image), take the result,

scale it down, scale it up...etc) to obtain a decent looking blur effect.

Let's begin with the code: surfaces are ideal to implement this. We will draw our sprite in a surface, scale it down, draw the scaled surface on a second surface, scale it up, then draw the second surface on the first one, and repeat the process.

First of all we need to initialize a few things in the creation event of the object we want to blur:

```
s = surface_create(sprite_width,
sprite_height); //main surface
saux =
surface_create(sprite_width*2,
sprite_height*2); //auxiliar surface

//clear the main surface:
surface_set_target(s);
draw_clear_alpha(c_black,0);

//clear the auxiliar surface:
surface_set_target(saux);
draw_clear_alpha(c_black,0);
surface_reset_target();

//enable linear interpolation:
texture_set_interpolation(true);

//the amount of blur we will apply:
blur_amount = 8;
```

Ok? So now we've got two surfaces ready to use, and linear interpolation

activated.

Now we need to make sure the memory assigned to the surfaces will be freed when destroying the object (destroy event):

```
surface_free(s);
surface_free(saux);
```

Now the cool part. We need to repeat a few times the scale down, scale up process in the step event:

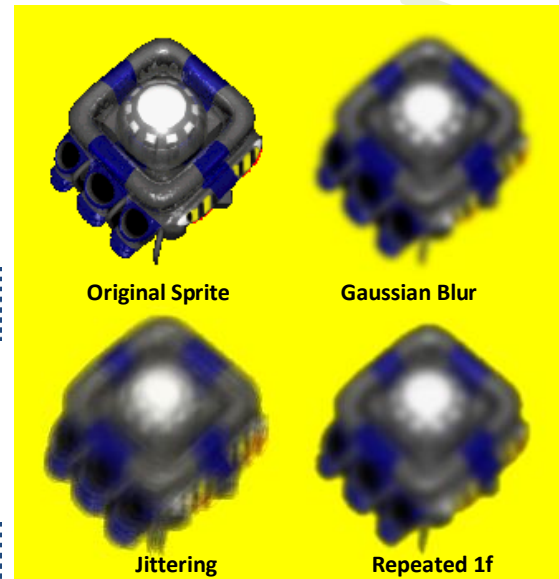
```
//clear the main surface and draw
the sprite there:

surface_set_target(s);
draw_clear_alpha(c_black,0);
draw_sprite(sprite_index,
image_index, sprite_xoffset,
sprite_yoffset);

/*scale down (to 0.5) and draw in
the auxiliar surface, then scale up
the auxiliar surface (to 2, because
0.5 * 2 = 1, the original image
size) and draw it in the main
surface again (thus blurring the
image due to linear filtering),
repeat.*/

repeat(blur_amount)
{
surface_set_target(saux);
draw_surface_ext(s, 0, 0, 0.5, 0.5,
0, c_white, 1);
surface_set_target(s);
draw_surface_ext(saux, 0, 0, 2, 2,
0, c_white,1);
}

surface_reset_target();
```



Finally, draw the main surface to the screen in the draw event:

```
draw_surface_ext(s, x-
sprite_xoffset, y-sprite_yoffset, 1,
1, image_angle, c_white, 1);
```

Execute it. As you can see, the quality is good. It almost looks like real gaussian blur. An higher blur value means less speed but the same quality. So this is ideal for very intense high-quality blur effects (if speed isn't that important).

Conclusion

Both methods work perfectly with rotated, color blended and animated sprites (although a little bit of "ghosting" takes place when using the repeated linear filtering method with animated sprites).

José María Méndez

Game Maker Affiliation Service

gmr

JessInc

Markup
MAGAZINE

GREEN'S
PUBLIC LIFE

Make use of this free service to exchange links with other GameMaker websites

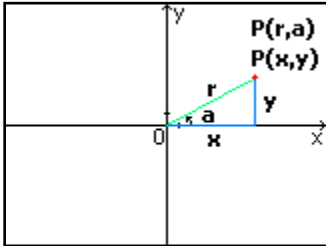
Over 20 GameMaker sites have already signed up

Video tutorials to guide to you through the process

Register Now!
CLICK HERE TO SIGNUP

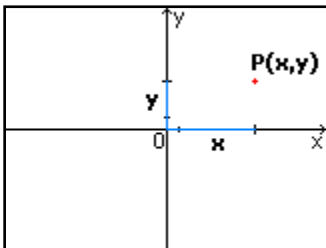
Cartesian & Polar Coordinates TUTORIALS

There are two ways to describe a point in a plane. The first way is by using Cartesian coordinates. The second way to describe a point is by using polar coordinates. The following image shows a point in the xy-plane, described both with polar and Cartesian coordinates:



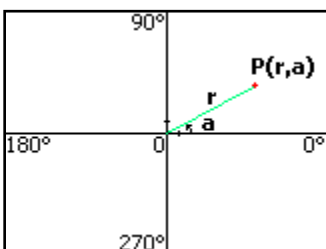
This article will explain how both coordinate systems are used, how to convert the coordinates from one system to the other and most important for Game Maker: explain the `lengthdir` functions.

Cartesian coordinates



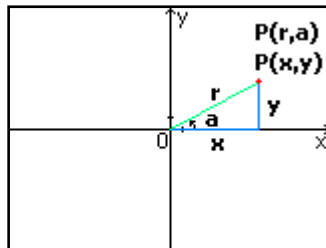
As you can see, the point is described with the coordinates `x` and `y`, where `x` is the distance from `P` to the y-axis and `y` is the distance from `P` to the x-axis. This way, we have fully described this point in the xy-plane.

Polar coordinates

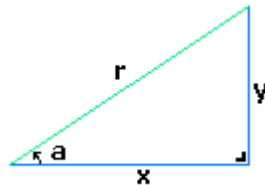


When describing a point with polar coordinates, we don't use the x and y-axis. In polar coordinates, only one axis is used, this is the 0°-axis or the "polar axis". 0 is the origin here. All distances are measured from this point. Rotation happens counter-clockwise. The point `P` can now be described by the length `r` (which is the distance from `P` to 0) and the angle `a` between `OP` and the 0°-axis. This way, we have also fully described point `P`.

Coordinate conversions



To convert Cartesian coordinates to polar coordinates and vice versa, we use an x- and y-axis. The distance `|OP|` is the length `r`. The angle `a` is the angle between the positive x-axis and line `OP`.

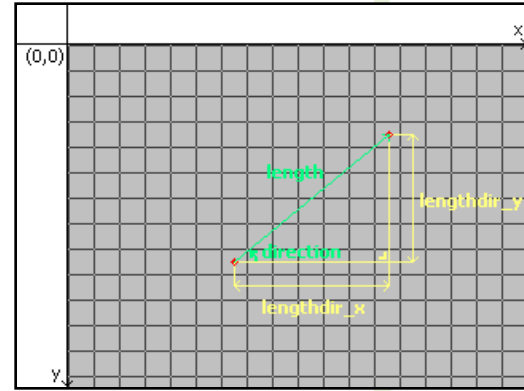


We can use the right triangle to solve the problem. As you can see,

$$\tan(a) = \frac{y}{x}, \text{ so that means}$$

$$a = \arctan\left(\frac{y}{x}\right).$$

This is a first formula that can be used to calculate `a` when `y` and `x` are known. When we use Pythagoras' theorem and apply it to the above triangle: $r^2 = \sqrt{x^2 + y^2}$. Now



we have two formulas that can be used to convert Cartesian coordinates to polar coordinates:

$$a = \arctan\left(\frac{y}{x}\right)$$

$$r^2 = \sqrt{x^2 + y^2}$$

Converting polar coordinates to Cartesian coordinates is even easier. As you can see:

$$x = r \cdot \cos(a)$$

$$y = r \cdot \sin(a)$$

And that's all you need to know for coordinate conversions.

The math behind lengthdir

The `lengthdir` functions are some very handy functions if you need to know the relative position between two points along a single axis (either the x-axis or y-axis). Something that is different in Game Maker, is that the y-axis points down: the y coordinate increases when going down. This means that a negative result means that the point lies higher in the room. The above image shows a room with 2 points in it.

The manual explains `lengthdir_x` and



Cartesian & Polar Coordinates TUTORIALS

`lengthdir_y` as follows:

`lengthdir_x(len,dir)` Returns the horizontal x-component of the vector determined by the indicated length and direction.

`lengthdir_y(len,dir)` Returns the vertical y-component of the vector determined by the indicated length and direction.

A very appropriate question could be: "I move length pixels in that direction, what are my x and y position now, relative to the x and y position I left from?" Then `lengthdir_x` and `lengthdir_y` would give you the respective answers. These functions do not necessarily return a positive value e.g. when the value returned by `lengthdir_x` is negative, the second point lies left of the first.

An example: you need to know the relative x position between two objects.

Let's say the first object is called `obj_1` and the second object is called `obj_2`. The distance between them is 100 pixels and the direction 60°. The following code will return this relative x position:

```
a=lengthdir_x(100,60);
```

The variable `a` now contains the x position of `obj_2` relative to the x position of `obj_1`. This result will be 50. Now why is this 50? To explain this, we need to go back to our right triangle. Note that also in the room, we can think of a right triangle, this time with sides length, `lengthdir_x` and `lengthdir_y`. Since it's a right triangle, the previously derived formulas are still valid. All we need to do is replace the variables:

```
r = length
a = direction
x = lengthdir_x(length,direction)
y = lengthdir_y(length,direction)
```

Then we get:

```
lengthdir_x(length,direction) = r *
cos(degtorad(direction))
lengthdir_y(length,direction) = r *
sin(degtorad(direction))
```

Direction is in degrees. When we go back to the example, we can now understand why the result is 50. The cosine of 60° is 0,5. Multiply this by `r`, which is 100 in this case, and the result is 50.

The two expressions derived above are completely equivalent.

And that's all there is to tell about Cartesian and polar coordinates. I hope this article has helped you understand these two coordinate systems.

Bart Teunis ■

Make your code easier to read

EDITORIALS

You may be perfectly happy coding in a haphazard manner however you should always ensure that your code can be easily read.

If you are working on a team project having code which can be easily read is vital. Otherwise when the game is being put together problems will inevitably occur when one member of the team isn't sure what a certain section of a script does. If you work alone you may think that this does not apply to you, but writing easy-to-read code will make your life much easier. It may take you a bit longer to write than your normal jumble of code but it will be worth it in the long run and could save you hours of frustration. For example with well-documented code you will be able to find the section you are working on quicker and if you take a break from coding you will be able to continue

where you left off without having to decipher what you wrote just two weeks previously.

If after release you come back to your completed project and want to fix some bugs that have been discovered, or decide to make a spin-off or more up-to-date version of your game you also need to be able to understand your previous work. Otherwise your existing code is as good as worthless. Here are some quick tips which, if followed, should make your code easier to read.

Use the tab key to indent your code. It's sitting there on the edge of your keyboard so use it!

Add comments. In Game Maker starting a line with `'/'` enables you to write a comment, visible only when you view the source code of your project. Use

comments to explain what your code is doing, separate sections of code and remind yourself what needs to be coded where.

Give your variables sensible names. At the time you are coding using variables called `d`, `f` & `g` may seem like a good idea and make perfect sense, but take a look at your project a couple of weeks later and you will be lost. Variables such as `playerhealth`, `topspeed` and `enemytype` may be longer but they will help reduce confusion, provided you can spell them correctly.

Blank space is your friend. Don't try to cram your program into as few lines as possible. Leaving a few blank lines is a good way to separate sections of your code.

Phil Gamble, GameMakerBlog.com ■

Binary for Beginners

It is my philosophy that the underlying principles of our numeral system should be taught early on to budding computer programmers as well as to novice mathematicians. Unfortunately, these fundamental concepts are often completely ignored in high schools and introductory programming courses all over the world and are often left completely unexplored until the college level. In this article I will attempt to illuminate some of the basic concepts behind positional notation focusing specifically on the binary numeral system (base 2) and its use in the computer sciences.

Positional Notation

Most readers will likely already know that today's world primarily uses a base 10 or "decimal" (also called denary [1]) numeral system which uses a total of ten unique symbols (0-9) to convey every possible real number. The most likely reason for this is that humans have a total of ten digits on both hands, lending to easy decimal finger calculations. Many readers, however, may not realize that the decimal system is not the only way of counting; in fact, it is merely one of an infinite number of numeral systems that are called "Positional" numeral systems.

To understand positional notation one must first know how our number system works. The positional number system allocates positions or places (eg. 1's place, 10's place, 100's place... etc.) that each number can fall into. Each position is related to the next by a common ratio called the radix or base. The decimal system is so named because it uses a base of ten, and thus the places are powers of ten (1, 10, 100,

1000, etc.). This means that the number 121_{10} can be broken down into $(1 \times 10^2) + (2 \times 10^1) + (1 \times 10^0)$ which is equivalent to: $(1 \times 100) + (2 \times 10) + 1$ (a 1 in the 100'ths place, a 2 in the 10'ths place and a 1 in the 1's place). Now do you see where the other positional numeral systems come in? Other numeral systems simply use a different base (and thus a different set of digits) to express the same set of real numbers. For instance, if we changed the base to 3 (3 digits, 0, 1 and 2), then 121_3 would be equivalent to 16 base 10 represented by: $(1 \times 3^2) + (2 \times 3^1) + (1 \times 3^0)$. I suggest that you make sure you have a firm grasp on this section before continuing on with the rest of this article.

Fractal Parts

Insofar we have only seen whole number representations of numbers. However, it would be pertinent to now mention the fractal part of a number. In any positional number system directly to the right of the one's place ($x \times b^0$) there is another place defined as ($x \times b^{-1}$) and another, ($x \times b^{-2}$) and so on and so forth. These negative places are often denoted by a separator between the whole and fractal parts of the number, in base 10 this separator is called the decimal point and is represented by a dot or comma (eg. 1.5 or 1,5). More generally this separator is called the "Radix point." Anything after the radix point is fractal, and anything before it is whole. This means that we could represent 3.14_{10} as: $(3 \times 10^0) + (1 \times 10^{-1}) + (4 \times 10^{-2})$.

Binary (Base Two)

The binary numeral system (base 2) uses only two digits - 0 and 1 - to reflect all real numbers. Binary is the most widely used number system in the computer sciences because it can be represented easily in hardware by the states of on (electricity in a circuit) or off (no, or less, electricity in a circuit). Nearly every modern electronic device uses binary as its native numeral system. Each 1 or 0 is known as a bit in binary (instead of a digit as in decimal) and it is common to call a bit of 1 true and 0 false.

Logical Operations

It is ever the goal of computer sciences to create computers that can make ever more complex decisions. The decision making process is accomplished on a PC by using logical operators on binary numbers, however, logical operators should not be confused with binary operations (that is, operators taking two arguments such as addition and subtraction which are relevant for any base).

Humans are forced to make decisions every day: we ask ourselves if we'd rather go outside OR stay inside, would we like to eat pie AND ice cream OR just pie? These simple decisions can be emulated by a computer using logical operators such as "And, Or, Not, and Xor."

The "and" operator simply returns true if both the left hand and right hand arguments are true. For instance, 1 and 1 is true while 1 and 0 is false.

The "or" operator does exactly what it



Binary for Beginners Cont.

sounds like as well: it returns true if one or the other argument is true (ex. 1 or 1 is true, 1 or 0 is true, and 0 or 0 is false).

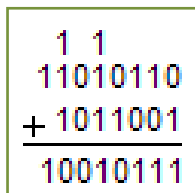
Xor you may not have heard of before as it is almost exclusively used in the computer sciences. Xor (eXclusive OR) returns true if one or the other of two arguments are true, but not if both are true. For instance, 1 xor 0 is 1, 0 xor 0 is 0 and 1 xor 1 is 0.

The "not" operator is a unary operation (that is, it takes one argument) which, once again, does exactly what it sounds like it does. Not 1 is 0, not 0 is 1, simple as that.

As we can see combinations of these operations can allow us to perform simple logic, if statement1 is true AND statement2 is true OR statement3 is true then... You get the idea.

Binary Operations

The simple binary operations we use every day (addition, subtraction, multiplication, etc.) can also be easily defined for the binary numeral system using simple logical operations. If a human were to wish to add two binary numbers he or she could simply add them like decimal numbers, that is:


$$\begin{array}{r} 11 \\ 11010110 \\ + 1011001 \\ \hline 10010111 \end{array}$$

However computers cannot reason through this method. Instead, a computer must use a series of logical operations

Hexadecimal (Base Sixteen)

The problem with binary is that it is very hard for humans to read quickly because even a small number can have a very large number of bits (ex. 100_{10} is equal to 1100100_2). *Ergo* it is often convenient to express binary numbers in a slightly different (but equivalent) way called hexadecimal. Hexadecimal, or base 16, is very easy for both humans and computers to read as it has a relatively large radix which is also a power of two making bin-hex conversions very quick on most computer processors. This means that instead of having to write 100_{10} as 1100100_2 we can simply write it as 64_{16} (notice how few digits that took). Numbers written in hexadecimal are often prefixed with 0x (# in HTML, \$ in GML). As you should now realize the hexadecimal numeral system will have 16 digits. This means that not only does it use the standard 0-9, but also the letters A-F where A = 10, B = 11 ... F = 15.

Storage

Despite the copious amounts of memory possessed by many modern computers numbers must still have some measure of consistency. In a plaintext file for instance every letter is stored as 8 bits which is also called a "byte" and translates into a two digit hexadecimal number. Two bytes (16 bits) make a "short" and 4 bytes (64 bits) make a "long." These values are fairly consistent on most processors, however, be warned: The byte is processor dependant! Not all computers use an 8 bit byte; in fact the size of bytes ranges from 5 to 12 bits! For our

purposes though we will be using the sizes defined above.

While storing data as a single byte is fine for text files, it has its limitations. Visualize a two digit number (base 10)... if we were to store all of our data as two digit radix 10 numbers we would find ourselves with a very large problem: namely, that a two digit base 10 number can only store a number as high as 99. Bytes are limited in the same way: they can only store numbers as large as FF_{16} or 255. Of course, we are not limited to two digits, therefore we can store numbers in shorts, longs, quads, and of nearly any other length (so long as it can fit into the memory). These numbers have higher and higher caps, just as a three digit base 10 number can store 999, a 64 bit long can store $FFFF\ FFFF_{16}$. For those of you who are familiar with the GM6 file format you will know that it stores most of its data as 64 bit longs for this very reason (a few values at the beginning of the file and some in resources are bytes).

Before we can begin to create our own files and store data there is one more thing we ought to know about numbers in general. We normally write numbers from right to left with the largest values on the left and the smallest on the right (ex. 100_{10}), however, to a computer it is often just as valid to write numbers the other way around (ex. 001_{10}) this is called endianness or "byte order" and is generally processor specific. The two most common byte orders are little-endian (little-end-first) and big-endian (big-end-first). You should now know enough to start reading and writing binary to and from files on your own!



Conclusion

Knowing how to use binary effectively will aid you tremendously in your game and software development. While we have only touched the tip of the iceberg this article will (hopefully) have given you enough knowledge to get you started in the wild world of binary and computer logic! There are many topics I did not cover here and wish I could have (bit masking, radix translations, shifting and rotating, etc.) but I'm sure that, equipped with this knowledge they will only be minor obstacles. Following are a few functions in GML to help you get started writing in binary.

Leif Greenman■

```
/*
** Usage:
**     file_bin_read_word(file,size,bigend)
**
** Arguments:
**     file      file id of an open binary file
**     size      size of the word in bytes
**     bigend    set to TRUE to use big-endian byte order (MSB first),
**              or FALSE to use little-endian byte order (LSB first)
**
** Returns:
**     an integer word of the given size from the given file
**
** GMLscripts.com
**/
{
    var file,size,bigend,value,i,b;
    file = argument0;
    size = argument1;
    bigend = argument2;
    value = 0;
    for (i=0; i<size; i+=1) {
        b[i] = file_bin_read_byte(file);
    }
    if (bigend) for (i=0; i<size; i+=1) value = value << 8 | b[i];
    else for (i=size-1; i>=0; i-=1) value = value << 8 | b[i];
    return value;
}
```

```
/*
** Usage:
**     file_bin_write_word(file,size,bigend,value)
**
** Arguments:
**     file      file id of an open binary file
**     size      size of the word in bytes
**     bigend    set to TRUE to use big-endian byte order (MSB first),
**              or FALSE to use little-endian byte order (LSB first)
**     value     integer value to write to the file
**
** Returns:
**     nothing
**
** GMLscripts.com
**/
{
    var file,size,bigend,value,i,b;
    file = argument0;
    size = argument1;
    bigend = argument2;
    value = argument3;
    for (i=0; i<size; i+=1) {
        b[i] = value & 255;
        value = value >> 8;
    }
    if (bigend) for (i=size-1; i>=0; i-=1) file_bin_write_byte(file,b[i]);
    else for (i=0; i<size; i+=1) file_bin_write_byte(file,b[i]);
}
```

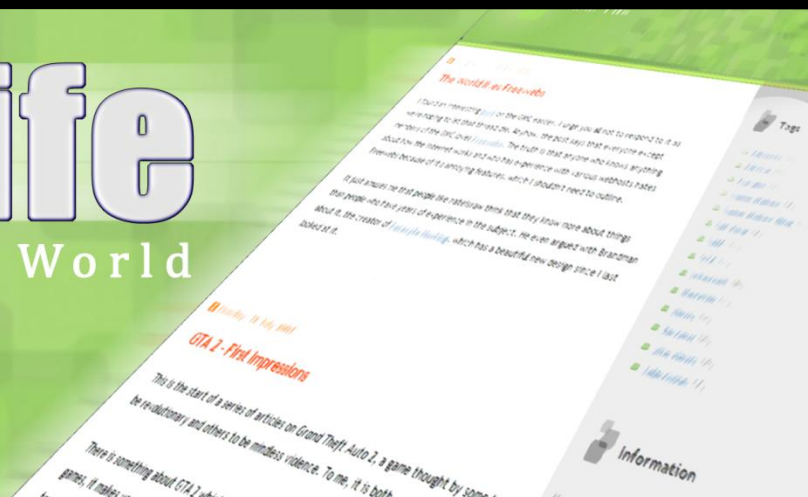
TUTORIALS

Stephan

REVIEWS

"Grego" Tyler ■

Your Connection to Our World



Modulo in Computer Science GAME MATH

Modulo is an important operation in both the computer sciences and in general mathematics. It allows for the use of complex logic in equations as well as loops and provides the ability to merge several equations into one formula. This paper deals with and outlines the use of the modulo operand in the computer sciences and provides several detailed examples of its use.

Modulo is a simple operand which can be defined quite simply as the fractal part of any ratio multiplied by the denominator of said ratio. It is taught first in elementary school, though not by its proper name, and all but forgotten in middle and high school. Its properties are never explained to the average student, its methods and means never comprehended. This paper aims to detail some of the uses of modulo and offer a general understanding and comprehension from which the hobbyist programmer and amateur mathematician can glean a fuller understanding of the mathematics and prose of a simple remainder.

What exactly is modulo? Modulo is the remainder of a division, just like you

practiced in elementary school, $5 \bmod 2$ is 1 because two goes into five two times with one left over, that remainder is the result of modulo. Because of this, modulo has several interesting properties:

1. $x \bmod y$ where $y > x = x$
2. $x \bmod x = 0$
3. $x \bmod 1$ where $x \in \mathbb{Z} = 0$
4. $x \bmod y = (x + (n * y)) \bmod y$ where $n \in \mathbb{Z}$

Property 4 leads modulo to often be associated with patterns that repeat themselves, lending it the name "clock arithmetic" after its use in the 12-hour clock system. For instance, $1 \bmod 12$ will equal 1, $12 \bmod 12$ will be 0 and $13 \bmod 12$ will also equal 1 and so on and so forth; because of this behavior, modulo can be thought of as a simple, logical, loop. Following from the previous example, let us say that we wished to write a function to convert from 24-hour time to 12-hour time. We might initially write down: $f(x) = x \bmod 12$, however, this function as we can see will return 0 for the inputs of 12 and 24. To remedy this, we may simply subtract and add 1 as follows: $f(x) = ((x - 1) \bmod 12) + 1$.

Now, consider the set of imaginary numbers: because the imaginary number i can be thought of as the square root of negative one, $f(x) = i^x$ has a very interesting solution set as illustrated in table 1:

Table 1

i^x	$f(x)$
i^0	1
i^1	i
i^2	-1
i^3	$-i$
i^4	1
i^5	i
i^6	-1
i^7	$-i$

As we can see, the values of $f(x)$ repeat themselves for every four values of x . This means that we can simplify the expression i^{294} down to i^2 using the equation $i^x = i^{x \bmod 4}$. Though this example may not be extremely practical from a programmer's perspective it is an ideal example of exactly what can be done with modulo.

Now let us look at a more practical

Autumnhouse
w • e • b d • e • s • i • g • n

25% off all websites
for GMC Members*

* terms & conditions apply

www.autumnhouseonline.com
webmaster@autumnhouseonline.com



Modulo in Computer Science GAME MATH

example. Let us say we are programming a small graphics editor and one of the features of this editor is a button which shifts all pixels to the right one pixel. We might simply add one to the x value of each pixel, but this will cause all pixels on the far right of the canvas to no longer be visible. Modulo to the rescue! Instead, we can use the simple equation: $x = (x + 1) \bmod \text{width}$ where width is the width of the drawing canvas in pixels. See how simple that is? The same might apply for looping a rocket to the other side of the screen in an asteroids style game, or when translating the contents of a data structure such as a grid (see Listing 1) or list.

Transformations are only one of the many uses of the modulo operator however. In an even more useful twist, modulo gives us an easy way to see if a number is even or odd (returning a Boolean value) by simply calculating the

number which we want to know modulo two since all even numbers are evenly divisible by two. And, because all integers are evenly divisible by one, we can check to see if a number has a fractal part by simply calculating the number in question mod 1 (if it returns 0 then the number is an integer). Modulo can be used in this manor to find a whole host of useful values including whether or not two numbers are powers of each other $\log_b x \bmod 1 == 0$ (just make sure to add a few error traps) and whether or not a number is evenly divisible by another number (as seen in the previous examples).

As you can see, modulo is an indispensable tool to the mathematician and programmer alike. Modulo pervades our culture and society, it is found in our system of time, in our measurement of angles, in the mathematics of our music, and in the

complex logic that makes most software possible. We as humans look for repetition and patterns in everything we do; we have set routines and schedules which repeat themselves regularly in accordance with a given timeframe and even the basic elements of our universe follow a simple periodic order. Modulo can be used to abstract many physical phenomenon and *ergo* is a cunning and indispensable device to have in your programming repertoire.

References

- Leijen, Daan (December 3, 2001). [Division and Modulus for Computer Scientists](#) (PDF). Retrieved on 2007-07-03.
- Multiple Authors. [Modulo operation](#). Retrieved on 2007-07-03.

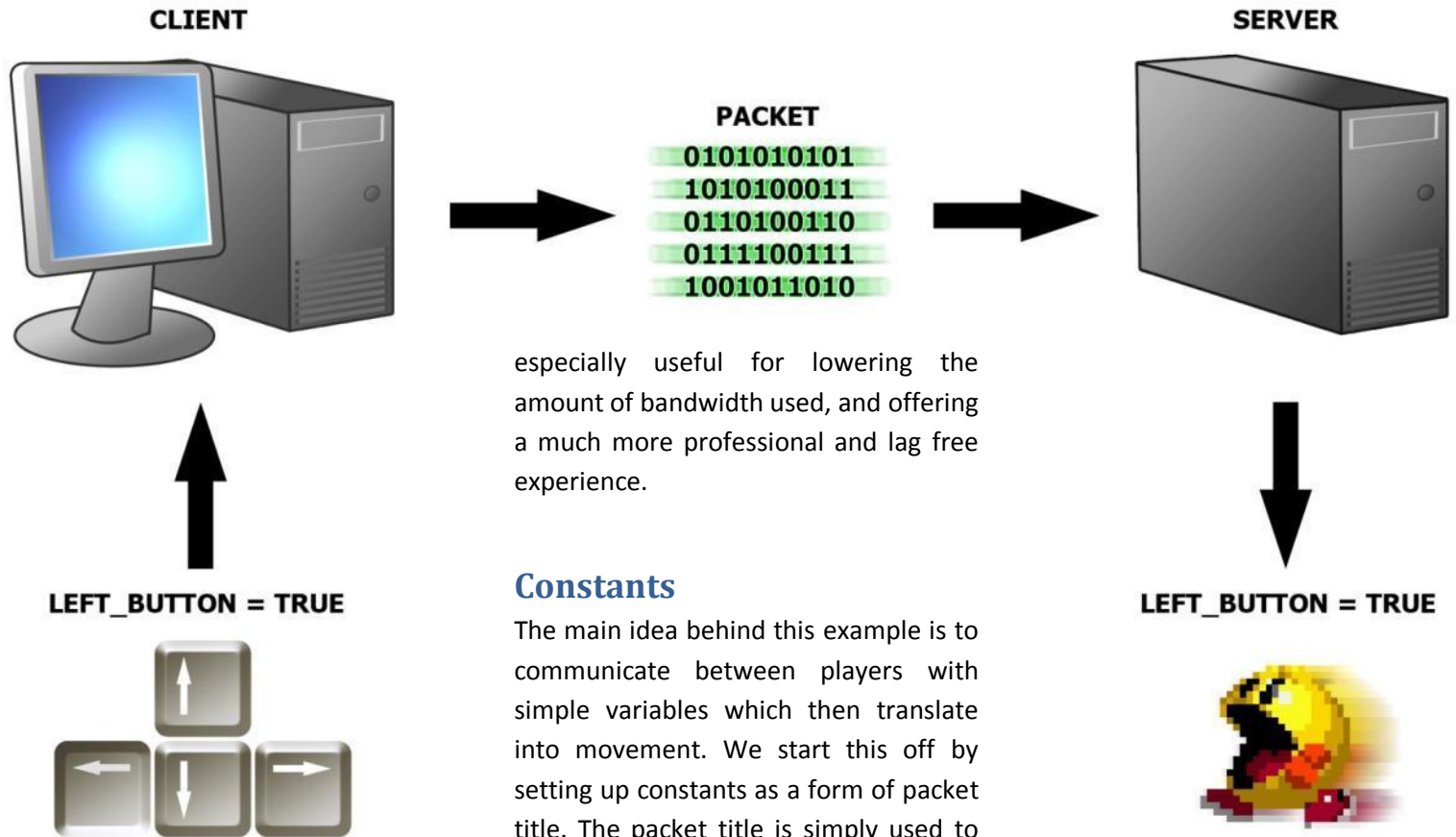
Leif Greenman ■

Listing 1: Translation of a grid using the Cartesian coordinate system

```
{
    // ds_grid_translate(dsid,horiz,vert)
    // This script is in the public domain and may be found at GMLScripts.com
    var dsid,w,h,sx,sy,mx,my,dx,dy,temp;
    dsid = argument0;
    w = ds_grid_width(dsid);
    h = ds_grid_height(dsid);
    sx = (((argument1 mod w)+w) mod w); // Notice the use of modulo
    sy = (((argument2 mod h)+h) mod h);
    mx = w-1;
    my = h-1;
    dx = mx-sx;
    dy = my-sy;
    temp = ds_grid_create(w,h);
    ds_grid_set_grid_region(temp,dsid,0,0,dx,dy,sx,sy); }
    if (sx>0) { ds_grid_set_grid_region(temp,dsid,dx+1,0,mx,dy,0,sy); }
    if (sy>0) { ds_grid_set_grid_region(temp,dsid,0,dy+1,dx,my,sx,0); }
    if ((sx>0) && (sy>0)) { ds_grid_set_grid_region(temp,dsid,dx+1,dy+1,mx,my,0,0); }
    ds_grid_copy(dsid,temp);
    ds_grid_destroy(temp);
}
```

Smooth Online Movement

MOVEMENT



especially useful for lowering the amount of bandwidth used, and offering a much more professional and lag free experience.

Constants

The main idea behind this example is to communicate between players with simple variables which then translate into movement. We start this off by setting up constants as a form of packet title. The packet title is simply used to give each packet it's own ID. By doing this, both client and server understand what data you are trying to send, and can read it accordingly.

Sending Data

Now that our constants/packet titles are set up, we can begin the process of sending data to the server depending on what buttons have been pressed. For example, if the player presses the left keyboard button, we send a packet to the server which basically tells it what button we're pressing. Afterwards, it updates our x and y position to re-sync the player. Now that the server is up-to-date on what the player is doing, it then forwards the received packet onto all clients connected to the server. These clients then do the same thing as the server by reading the data sent.

Transforming Data into Movement

Now that we are successfully sending packets to the server and clients that

Dragger DLL

The dragger DLL allows windows to be dropped inside a Game Maker window. A single file or multiple files could be simultaneously dragged into a Game Maker window. The DLL can figure out the number of files being dropped into the window by using an index (0-based). The DLL download comes with an example in a GM6 format.

Get it now!

<http://xrl.us/draggerdll>

Introduction

The other day I was browsing the Game Maker Community, testing some of the online games and engines made with Game Maker. A quick look at most of the games and engines being coded helped me come to the conclusion that the community as a whole is still very new to the world of online programming. This is made evident in part by the lack luster games being programmed that make little to no use of a widely used technique called Dead Reckoning. This technique is used to save on bandwidth and lag in an attempt to guess what certain info is depending on data already received. The example I am about to talk to you about is a form of Dead Reckoning. It calculates the movement and fluidity of a player depending on certain data that has already been received. This is

Smooth Online Movement

MOVEMENT

contain the info to what button the player is pressing, we can begin the process of transforming these variables into simple movement of the player. An example of this in step-by-step order:

1. Player presses left keyboard button
2. Packet is sent containing info for left keyboard button
3. Packet is received and read in the form of simple variables
4. The game checks those variables and translates them into movement

As you can see, it is a pretty simple process. One in which only a little common sense is required to create a highly used form of Dead reckoning.

Physics

Unlike in online games where player movement is done by constantly updating the x and y position of the player, we must now program in some simple physics for the dummy player so that he acts in a manner that mimics that of the actual players key presses. We do this by giving him some collision

Online Movement Sending X,Y



Online Movement Sending User Input



checking, gravity, etc. This is a fairly easy part; just remember to make these physics exactly the same as the ones affecting your own player. If not, it can create quite a bit of lag.

The End

Hopefully by reading this little summary, you now have a decent insight on what it takes to program a more professional online game. If you have any questions regarding the technique or the example, please e-mail me at jakethtsnake3636@hotmail.com. For a look at a full-fledged MMO using this technique, please visit nightfallonline.co.nr or stick-online.com.

Jake

EDITORIALS

Promoting your Game to the GMC

To start off with I will clarify that in this article the "Game Maker Community" refers to people who use, or are familiar with, Game Maker and **not** the name of the official forums. It is of course possible to promote your game outside of the community, and this is something which more and more people are beginning to do, however this article is only about promoting 'internally'.

Forum signatures are a great place to advertise since it is essentially free space, and if you are an active poster it will quickly spread your message across many different topics.

You can obviously start a topic promoting your game in the Game Maker creations section of the official forums, and you could also post your website in the Website Announcements section to coincide with your release. There isn't just the one forum though, if you are a member of any other forums which focus on Game Maker or game development

you should also try the same there. For example many Game Maker 'teams' have their own forums, you can easily copy and paste the same post onto a number of these, although I don't condone spamming of any kind.

Uploading at community based sites such as 64digits won't be a bad thing either, nor will posting your game at YoYo Games, where there is big audience.

Affiliating (link exchange) with other sites is a good idea but you only get a small canvas on which to show your message by using this method. If your banner will appear at the bottom of a page alongside hundreds of others this will also be pretty useless (says the man who runs a [Game Maker affiliation service](#)). Textual links may be more beneficial, particularly when it comes to Search Engine Optimization, however this shouldn't be your primary concern when promoting to the community.

I believe it would be more valuable to try

to get your game reviewed or previewed by one or, even better, both of the Game Maker magazines. Provided you have put a sensible amount of time into your game your review should in effect be free advertising and you will also get some constructive criticism on top of that. These magazines also give free advertising in exchange for articles so if you get writing about something Game Maker related, not just shameless promotion, and submit it to one of them you will probably be given a quarter page ad or something similar. From my experience ads in Markup do get clicked and do get results.

As well as choosing the right place to advertise your message also has to be effective, you will only have a limited amount of space so you need to make the most of it and make sure you have got everything right. Sloppy spelling and a misspelt URL won't do you any favors.

Phil Gamble,

GameMakerBlog.com

Introduction to DLL-making

Along time ago in a land far, far away, there was a programmer. This programmer had a project that had many, many functions which needed to be shared between several source files. He toiled away, copying and pasting every time he made a slight change to the functions in one of his programs. Finally he decided enough was enough and a brilliant idea came to him: thus the DLL was born! Before we start, I must warn you, DLL's are not for the faint of heart, to understand this tutorial you must be familiar not only with GML, but you must have some general C++ knowledge as well. If this has not deterred you, read on!

What is a DLL

The Dynamic Link Library, or DLL, is a collection of pre-compiled code that can be linked with and executed dynamically at runtime (as opposed to a static library which can only be linked to at compile time). DLL's are one of the programmer's most useful tools; they allow for easy management of shared code between many applications and allow you to update code in one spot and one spot only without having to recompile your entire project (See Figure 1).

The Basics

Game Maker supports limited linking with DLL's compiled in another language such as C++ or Delphi. DLL's have several advantages over GML code: first of all they are compiled; this lends them to much faster execution speeds than scripted GML code. Secondly, you can do a lot more in a real programming

language which means that DLL's can add functionality that was previously impossible to Game Maker. Though DLL's can be written in a variety of different programming languages, in this article we will be focusing on creating DLL's in C++. The concepts covered in this article however should apply to most any language in which DLL creation is supported. What follows in Table 1 is a detailed comparison of GML code as opposed to compiled C++ code. This highlights some of the advantages and disadvantages of using DLL's as opposed to standard GML scripting. If you want to skip these small technical details and jump strait to programming your own DLL you may want to skip down to "Programming DLLs" on the next page. There are of course some discrepancies in the table: dynamic typing is not necessarily a pro for GM, but for our purposes it will be, also ease of use and learning curve are the opinion of the author.

Data Types

Now down to business. As you can see, DLL's offer several advantages over C++

code. One of the many advantages of using DLL's is that C++ has many more data types than GML and can define custom data types which are just as valid (to C++) as the standard types. Unfortunately, GM only supports two of the data types which C++ is capable of: Null terminating strings, and real numbers (doubles). Now, when I say GM only supports these data types I mean just that. Your DLL can define its own data types or use as many others as it wishes, however, GM can only supply the DLL with doubles and strings, and the DLL can only return doubles and strings to GM. Let's take a look at each of these types and see exactly what kind of data you can store with them.

Doubles, or real numbers, are signed 64 bit double precision floating-point numbers that can store values in the range of $1.2 \times 10^{\pm 4932}$ (19 digits) and have a precision of about 8 bytes (on most computers). They are initialized in C++ using the keyword "double".

Null terminating strings on the other hand are lists of characters that only end when the escape sequence `"\0"` (null) is reached. I prefer to use the LPSTR class found in the standard windows includes (`windows.h`).

Programming DLLs

Open up your favorite editor, prepare your favorite compiler or GUI, and let's get started writing a DLL for Game Maker! Create a new project or file and set up your compiler to compile a DLL (this is compiler specific, so you'll need to read the documentation if you don't know how to do this). Now let's create a simple function to add two numbers in

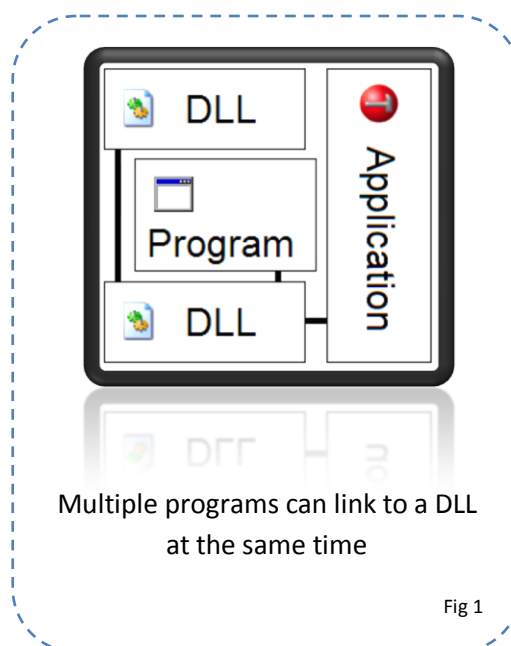


Fig 1



Introduction to DLL-making

TUTORIALS

Table 1	GML	C++
Dynamic Typing	Yes	No
Compiled	No	Yes
Flexibility	Worse	Better
Speed	Slower	Faster
Dynamic Resource Access	Yes	Not Allowed ¹
Memory Pointers	No ¹	Yes
Ease of Use	Easier	Harder
Custom Data Types	No	Yes
Learning Curve	Smaller	Steeper

C++. We will call this function `dAddNum` and it might look something like this (remember that we can only use doubles and strings for arguments and return types):

```
double dAddNum(double dNum1, double dNum2)
{
    return (dNum1 + dNum2);
}
```

There is still one problem though, if we compile a DLL and try to execute this function from Game Maker it will not work. This is because we have not told the DLL to export the function to Game Maker. To do this we must add the following to the beginning of our function right before the declaration:

```
extern "C" __declspec( dllexport )
```

Rather than add this before every function, however, I prefer to define a single keyword that will act as a stand in for this line of code. This makes our final addition function look something like the one found in Listing 2.

Strings on the other hand are a little more complicated to pass and return. C++ does not define a built in class for storing strings, instead we have to use either an array of bytes (`char`) or one of the many string classes found in the

standard C++ includes. I like the `LPSTR` class which can be used by simply including `windows.h` with your DLL. That's about it for defining functions from within DLL's! Of course, you can do much more advanced things with your functions than just perform basic calculations (which are more suited to GML).

GM supplies a certain function that allows your DLL's to deal with the game window directly. This function is called `window_handle()` and (counter intuitively) returns the handle of the game window. This handle may be used in your DLL's to modify the windows position, attributes, contents, styling, etc. However, this is not a C++ tutorial, and we will not be covering the means and methods of window manipulation here; for an example see the code example at the end of the article. The final topic we must cover before we can really be expert DLL creators is that of calling conventions. Game Maker supports two calling conventions: `cdecl` and `stdcall`. To understand the difference between these two methods and what they do, we must first know a little bit about how C++ calls its functions. C++ passes all arguments to

functions to a block of memory called the "stack." The stack is a low-level data structure that functions exactly as you would expect a stack to, when you call a function its arguments are put (or "pushed") on the top of the stack in reverse order. When the function returns, the arguments are "popped" from the top of the stack on a first-in-first-out basis. This means that whatever the last thing you placed on the stack was will be the first thing to come back off of the stack.

`Cdecl` is the default calling convention for C programs and global functions in C++ programs. When the calling convention is set to `cdecl` the caller (Your game) is responsible for popping data from the top of the stack. This is all done automatically and you probably don't need to worry about it.

`Stdcall` on the other hand requires that the callee pop arguments from the stack. This is the default convention for the Win32 API. Because the callee is responsible for removing its arguments the `stdcall` method is slightly faster, however, it prevents the use of the `...` operator for sending runtime defined arguments.

If you're uncomfortable choosing a calling type, better make it `cdecl`.

Using the DLL

Now finally comes the fun part: implementation, actually using our newly created DLL in Game Maker! Before GM7 we had to use a series of functions to initialize the DLL, define the functions, and call the functions. GM7, however, does away with all that and



Introduction to DLL-making

gives us a single mechanism for the inclusion of a variety of file types: The GEX file.

The GEX file will act as a container for our DLL file that will export it to a temporary directory at the start of the game, call any initialization scripts that need to be called, define every function within the DLL, and finally, when everything is all done, call any tear down functions and release the DLL from memory. Unfortunately, you cannot create extensions from within Game Maker itself. To create your extension you will need the free utility found [here](#). From within the extension maker you can simply click "Add DLL" to import your DLL and then define your functions by adding them to the functions list (see Figure 2). Now, on the file menu, click "Build Package..." to compile your project into a single GEX

file which can be imported into Game Maker. To execute your DLL's functions all you have to do is call the name of the function or the alternate name you defined in the extension builder from within Game Maker!

Conclusion

You should now know enough about DLL creation to get out there and start creating some basic DLL's! Remember though, DLL creation is a very advanced form of programming which can be hard to debug. Before you start trying to make your own GMSoc or Xtreme3D, make sure you know enough about calling conventions, memory allocation, and data types to prevent memory leaks and buffer overflows which can lead to security vulnerabilities in your games. Below is a list of resources which you may find useful when developing your

DLL's, good luck!

Resources

- [Microsoft Visual C++ Express](#)
- [Visual C++ Developers Center](#)
- [Dev-C++](#)
- [MinGW](#)
- [DLL Programming Resources on the GMC](#)
- [C++ for Dummies](#)
- [Game Programming Gems](#)

Footnotes

¹It is unfair to say that the inability of C++ to access GM's resources is a con. After all, this is the fault of Game Maker, not C++

²GM does not define pointers but they can be emulated for resources.

Leif Greenman ■



```
#define export extern "C" __declspec( dllexport ) /* __stdcall if we will cleanup */
#include <windows.h>
#define MAIN_EDIT 101 // Just an edit ID
export double dAddNum(double dNum1, double dNum2)
{
    return (dNum1 + dNum2);
}

export double dTextBox( double dX, double dY, double dWidth, double dHeight, double dWinHandle)
{
    HWND hwnd = (HWND)(int)dWinHandle;
    CreateWindowEx( WS_EX_CLIENTEDGE,
        "EDIT",
        "Hello World",
        WS_CHILD |WS_VISIBLE |WS_VSCROLL |WS_HSCROLL |ES_AUTOHSCROLL |ES_MULTILINE,
        (int)dX,
        (int)dY,
        (int)dWidth,
        (int)dHeight,
        hwnd,
        ( HMENU ) MAIN_EDIT,
        GetModuleHandle( NULL ),
        NULL );

    return (0);
}
```

Level Editors pt. 1: Format

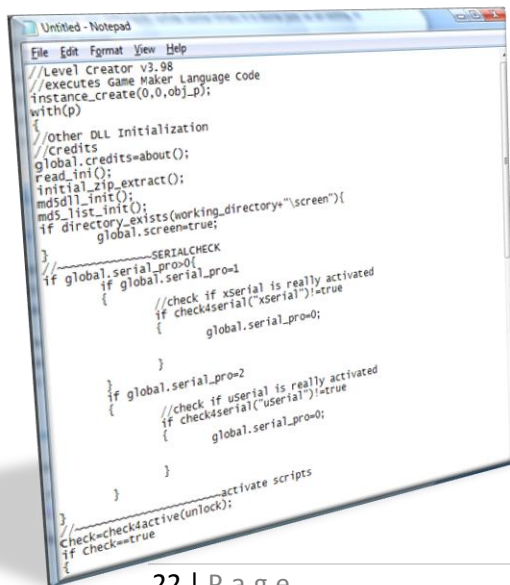
A level editor in a game is considered a massive feature by many, and indeed it is. The reason developers add level editors to their games are different: while some times it is done just as an extra, in other times there could be a compelling reason.

Take for example my most recent project, which involves a crime scene investigator trying to solve cases – different cases are represented as different levels and in order for my game to be successful; those cases need to be really interesting: and that's where I fail! That's why I decided to create a level editor so that others could make interesting levels for the game, and later packed as true levels.

Methods of Creating Level Editors

METHOD 1: The famous `execute_file()`

The most insecure of all methods on earth is `execute_file`! What happens is that the code of creation of each object is directly added as GML code to a file – this includes creating different instances



of objects, setting variables for these objectives, etc.

As it will be discovered later, the method isn't so flexible, might need a lot of hard-coding (which means it would make it difficult to update), and is simply completely insecure – someone could make a level that deletes your system files! Nevertheless, we will briefly discuss the different ways to create such a method.

By using the WITH statement

A great function to do this would be a `with(all)` statement. In that statement, something such as:

```
code+="instance_create("+string(x)+",\n"+string(y)+",\n"+string(object_name(object_index))+",\n");";
```

To set more variables to the created object, `obj=` could be added before the code, to create something similar to this:

```
code+="obj=instance_create("+string(x)+", "+string(y)+",\n"+string(object_name(object_index))+",\n");";
```

Now, to define variables such as `my_life` for that objects, a simple line of code could be used:

```
obj.my_life=79;
```

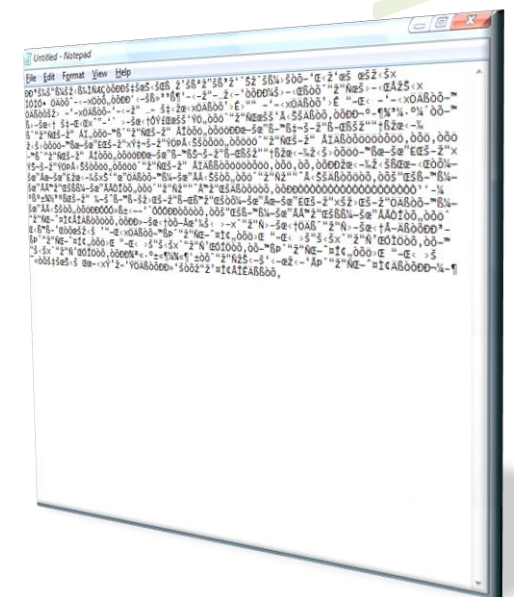
The `with` statement could also include a `switch()` statement, where the `object_index` of each object is tested – and accordingly, different types of variables could be added to the object.

Embedding code directly into objects

Another alternative could be to directly add code to an object's create event – so

that whenever it is created in the level editor, it adds some GML code to a global variable about its creation, position, etc. While this could prove more flexible at some times than using the `with` statement, it also creates a problem: destroying the object. However, using string replacement functions, or by simply adding a `with(...){instance_destroy();}` line of code could solve that problem.

METHOD 2: A relatively more



secure alternative to `execute_file`

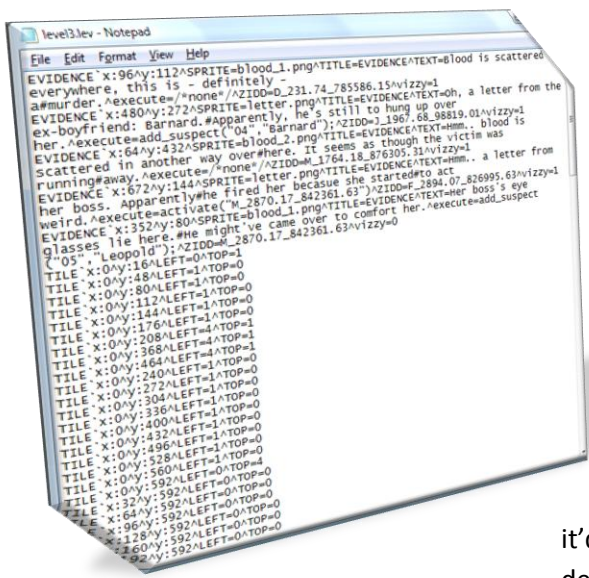
Instead of executing a file, how about executing a string? Sounds stupid, but a file with the same type of GML code as shown above could be encrypted, read by the actual game, stored in a string, decrypted, and executed.

That could make it safer, but decryption has been proven to be a weak form of protection – especially when malicious coding could still be injected in the file after the encryption has been cracked.



Level Editors pt. 1: Format

TUTORIALS



METHOD 3: A special file format

There is no doubt making a special file format is the most secure and efficient method of storing level data. The idea is that values are stored in specific locations, and are separated by different separator characters (example: CSV). This way, the game could read a value, either keep it as a string or turn it to a real value using the `real()` function – and setting a variable to that value.

This is the method I personally use for my project, and this is the method we will further discuss in this article.

The format

We could take advantage of Game Maker's ability to read only one time at a row. Different pieces of data such as different objects to be added to the room will be stored in separate lines – this means each `file_text_read_string()` carried out would read a single line, and therefore a single object.

Since most games have different types of objects, each with different variables

that need to be defined, it is a good idea to include the object's name as the first part of each row, to define which object is being created.

For example, a certain object might only need two variables defined, while others might need several ones – that could introduce array index out of bounds or other types of errors so

it'd be better if object names are defined at first.

Character Separated Values vs. Fixed Length Record

On one hand, reading and separating Character Separated Values is not a native function in GM, and coding such thing in GML could be difficult – and at the same time, Fixed Length handling is easy in Game Maker, as functions like `string_char_at()` and `string_copy()` could easily create parts of a string that lies within a certain position.

However, using fixed length records mean that there should be a maximum length for all records, and it also means that the file size would be much larger than variable-length records.

Character Separated Values succeed in every way when compared to Fixed Length Records except for the fact that they cannot be natively and directly used in GM. However, GMLscripts.com has come to the rescue – as shown in Issue 4, script of the month – and gave us the `explode_string` function. Which separates a string into multiple strings in an array.

Separators

Though the use of commas, colons, or semi-colons is generally considered okay, many variables might store text – and text generally contains all these. For that reason, I suggest to use an uncommon character as the separators, I use `(')`, `(^)`, `(|)`, etc.

To make things “nicer”, I often use two separators: one to separate the object name from the rest of the variables, and the other to separate the different variables from each others.

Here's an example of what a file could look like:

```
EVIDENCE`96^336^blood_2.png^Evidence
^Sample evidence text.^/*none*/

EVIDENCE`96^144^blood_2.png^Evidence
^Sample evidence text.^/*none*/

TITLE`128^144^0^0

TITLE`160^144^0^0

TITLE`192^144^0^0
```

Reading the Format

Using two separators means that readings could occur on multiple stages. In the first stage, the strings are

AStar DLL

The AStar DLL is one of several path finding DLLs based on the effective A* algorithm. It allows for fast real-time path finding, so that the object could move around blocks and other restricted areas (obstacles) to reach its target position.

It is very effective, and allows for awesome AI!

Get it now!

<http://xrl.us/astar>

Level Editors pt. 1: Format

Table 1: First stage

SPR[0]	SPR[1]
EVIDENCE	96^336^blood_2.png^Evidence^Sample evidence text.^/*none*/
EVIDENCE	96^144^blood_1.png^Evidence^Sample evidence text.^/*none*/
TILE	128^144^0^0
TILE	160^144^0^0
TILE	192^144^0^0

Table 2: Second stage

PART[0]	PART[1]	PART[2]	PART[3]	PART[4]	PART[5]
96	336	blood_2.png	Evidence	Sample evidence text.	/*none*/
96	144	blood_1.png	Evidence	Sample evidence text.	/*none*/
128	144	0	0		
160	144	0	0		
192	144	0	0		

separated into 2x1 arrays.

The first `explode_string()` function will separate the object name from its variables (table 1, overleaf).

Now a tile only needs to have 4 variables read, while the evidence object needs six. So, a switch statement is done, if the variable `SPR[0]` is "EVIDENCE" all 6 indices of the array are read, but if `SPR[0]` was "TILE", the final two indices are not read – as reading them causes an error.

A second `explode_string()` function is

performed, this time to separate different variables from each other (table 2).

Writing the format

The act of writing the format by the level editor isn't hard anymore. `with()` statements could be carried out, but instead of using `all` we use a specific type of object each time. So, we'll have a `with(obj_tile)` and a `with(obj_evidence)`, each of them adds the name of the object, along with character separated variable values.

Conclusion

In this issue, you learnt about the different ways of storing information about levels. Next issue, we'll talk about the different methods of creating a level editor's interface.

This covers the separation of the items of the level from the actual interface, ways of making a good interface, etc.

Eyas Sharaiha ■

Cage Match no more.. again?

N E W S

Though not directly related to development – the Game Maker Community's Cagematch (once seen as a method of showcasing one's work) – has now been suspended until further notice.

The Cagematch used to be run by Dex in the old days, until he became too busy to handle it, and eventually the Cagematch ceased to exist.

Not-so-long-ago, Ablach Blackrat took over, and revived the long-liked tradition

of a weekly Cagematch – until recently.

The Cagematch was suspended after about 100 mysterious votes came from nowhere to one of the games. However, the creator of the game insists he has nothing to do with it.

It now appears that a hidden iframe or a similar trick in a certain site was placed, which forced users to automatically vote for that certain game.

Following these events, the Game Maker Community moderation staff decided to suspend the Cagematch until further notice.

It is not known whether the Cagematch will reopen in the near future, and it is also not known whether it has only been suspended for inspections or not.

Eyas Sharaiha ■

To start, let's say that you've made a game. Someone plays it, and complains about the incorrectness of the physics in a certain part of the game. As a developer, how can you ensure that all the physics in your games is accurate, which also aids you in making some pretty cool games too? The answer can come to you in a single package, GMPhysics.

Introduction

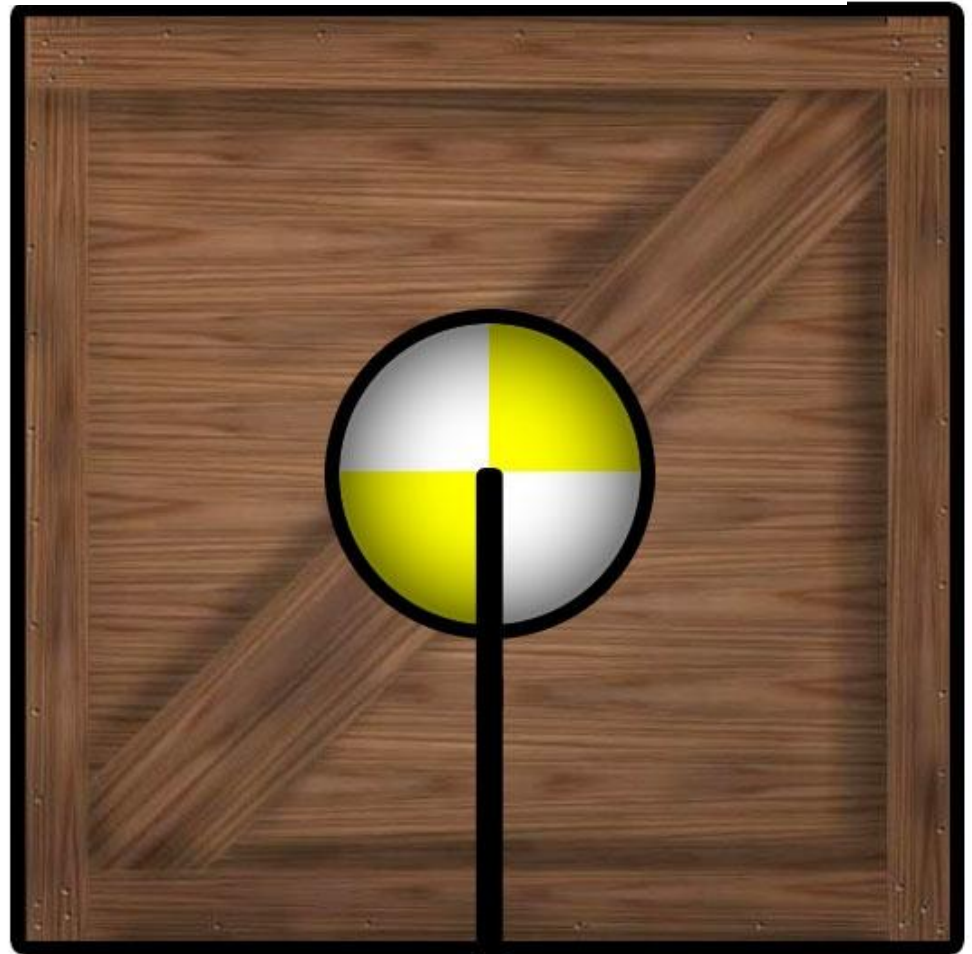
So, when you open it up, you see a bunch of DLLs and a couple of example games. Look through them, and when you're done playing, open up the platform tutorial included.

Before we start making physics, we're going to have to learn about how it works. Clear the room of everything but the control object (it should be displayed as a blue question mark.) Then, open up the control object in the object editor. Double click on the code in the Create event. Something like this should pop up:

```
{  
  init_physics();  
  create_body(0,room_height,STATIC,SHAPE_PLANE,90);  
}
```

So, let me explain this. `init_physics()`; simply initializes physics to run. The next piece of code is a bit more complicated. We'll take it step by step. The function, `create_body()`, is pretty self-explanatory. The first two arguments are x and y. So, this creates a body at the bottom-left corner of the room. The next argument is the density. This can be any positive number (for a good scale, use 1 as a small chapter book, 3 as a dictionary and 5 as a

medium-sized rock.) But think of it this way. If the body is just sitting on the bottom of the screen, it will fall off. But, if you put `STATIC` in its density, it becomes ground. Now, nothing will affect it, but other things will be affected by it. The next argument is `SHAPE_PLANE`, which indicates that the shape is a plane, and the last argument is the argument specific to that shape, which is direction. Unlike Game Maker, the directions are as follows: 0° is up, 90° is right, 180° is down, and 270° is left. Because the plane is on the left side of the screen, we want it to stretch until it reaches the other side, the right. Different shapes have different arguments, too. For example, a ball's only argument is its radius, and a rectangle's two arguments are its width and height.



In every step, you'll want to update the simulation. So, open up the code in the step event and you should see the following:

Easy Inventory

"Easy Inventory" is a set of Game Maker scripts designed to allow you – the developer – to add inventories to your game. This is particularly useful in RPG games like Diablo, etc.

It allow for the drawing and manipulation of the inventory and the items in it using simple lines of code.

Get it now!

<http://xrl.us/inventory>



```
{  
    update_bodies();  
}
```

This updates it every step, and this is very important.

The last thing you'll need to look at is the Game End event. In it should be something like the following:

```
{  
    release_physics();  
}
```

That simply gets rid of the physics. You don't want it to be calculating nothing after you quit, do you?

Well, that's it. You now know how to make a static body and maintain your physics correctly. Now here comes the fun part: making all those bouncing balls and falling blocks.

Making Bodies

So, create a new object called `obj_box` and give it a simple 32x32 box sprite. Make sure that the origin of the sprite is in its center! This is very important!

Put a Create event in it, then put the following piece of code in its create event:

```
{  
    h = create_body(x, y, 1, SHAPE_BOX,  
        32, 32);  
}
```

From what you've learned before, this shouldn't be too hard. It creates a box-shaped object with a width and height of 32 (hence the sprite.) It creates it at its x and y, and gives it a lightweight density of 1 (you can change this, but don't make it past 10. It'll be too heavy.) We assign it to a variable so we can easily access its physical body later.

Before it does anything, you also need to make it update itself. So in the Step event, put in this:

```
{  
    object_update(h);  
}
```

That simply updates it. The last thing you'll want to do is make it so that when its Game Maker instance is destroyed, the physical body will get destroyed too. So, make a Destroy event, and in it put:

```
{  
    destroy_body(h);  
}
```

Now, go back to the controller object. Create a mouse Global Left Pressed event, and in it put the following piece of code:

```
{  
    instance_create(mouse_x, mouse_y,  
        obj_box);  
}
```

That makes an instance of object box at the mouse x and y, you should be familiar with this if you have coded before.

Now, run the game and click anywhere. If you haven't made any mistakes, it's quite fun, isn't it?

Making Joints

Joints are pretty easy to make, and can provide you with pretty amazing results.

There are three types of joints, which will all be explained.

The first type is a fixed joint. Fixed joints



QUICK REVIEW

D3D9 Wrapper

This Game Maker DLL is a wrapper of the latest version of DirectX 9's Direct3D library. Game Maker itself incorporates features such as rotation, etc. from DirectX 8 – but doesn't even use it to its full potential. This DLL – a work in progress – seems as if it will become faster, and give us some interesting features in the near future. A GMK example is included.

Get it now!

<http://xrl.us/d3d9w>





are very easy to create, and link two objects together. To make one, use this:

```
{  
create_joint(something.h,  
something.h, JOINT_FIXED);  
}
```

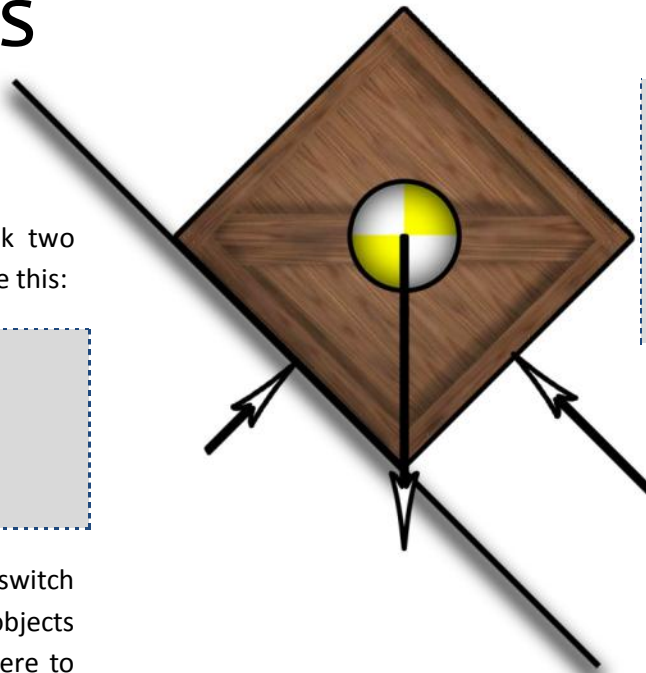
To make the joint work correctly, switch the two 'somethings' to the objects you want to join. The "h"s are there to make sure it links the bodies and not the instances. For example, if you wanted to link together our `obj_box` and another `obj_ball`, you would use:

```
{  
create_joint(obj_box.h,obj_ball.h,JO  
INT_FIXED);  
}
```

Note that you have to be careful when using object names. Creating more of that kind of instance could mess it up. Remember that you can assign instances to variables, too. Kind of like this:

```
{  
firstvar = instance_create(100, 100,  
obj_box);  
secondvar = instance_create(200,  
200, obj_box);  
create_joint(firstvar.h,  
secondvar.h, JOINT_FIXED);  
}
```

Take note that this assumes `firstvar` and `secondvar` have already created their bodies. If you put a `create_body();`



```
{  
create_joint(firstvar.h,  
secondvar.h, JOINT_SLIDER,  
get_body_x(firstvar.h),  
get_body_y(firstvar.h), 180);  
}
```

Everything from the hinge joint is included. The last argument, which is new, is the direction in which the slider joint points.

Conclusion

Well, that's it. You now know how to create and maintain physics, create static bodies, create bodies, and create joints. Of course, this isn't the end of GMPhysics. It can be used to create rays, water, springs, wind, magnetism, soft bodies, and a whole lot more. Here's a hint: See what happens when you create a bunch of spheres and link them all together from 1 to 2, 2 to 3, 3 to 4, and so on.

Sean Flanagan ■

event in their create event, it should work fine.

The next kind of joint is a hinge joint. Hinge joints are slightly harder to understand.

To create one, use the following piece of code:

```
{  
create_joint(firstvar.h,  
secondvar.h, JOINT_HINGE,  
get_body_x(firstvar.h),  
get_body_y(firstvar.h));  
}
```

The first three arguments link the two bodies with a hinge joint. The last two is where the joint is anchored. If you want B to swing around A, anchor the first one. If you want A to swing around B, anchor the second. (Usually this won't matter unless you're using a static body as one of the objects.)

The last kind of joint is a slider joint. Slider joints are usually the hardest type of joints to understand. Here's an example of the code used for creating a slider joint:

Dialogue System

This dialogue system imitates the one present in Baldur's Gate. This includes scrollable dialogues, with the ability to use the script in conjunction with room views (early versions require editing, latest version has it native).

Features include multi-line with setting line limits and conjunction with scrollbars and scrolling buttons.

Get it now!

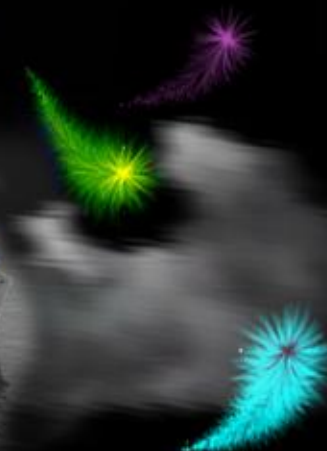
<http://xrl.us/dialogue>

GreenMan Games

```

var sprite,a,w,h,n,xoff,yoff,surface,i;
sprite = argument0;
a = draw
w = s
h = s
n = s
xoff = s
yoff = s
surface = s
surface_
draw_clear(c_black);
draw_set_blend_mode_cat(bm_one,bm_zero);
draw_set_alpha(1);
for(i=0; i<n; i+=1) {
    draw
}
draw
draw_
return (string(surface) + ':' + string(sprite));
}

```



Leif Greenman

Need I Say More?

<http://www.greenmangames.vze.com/>



Extension of the Month

EXTENSION

Powered By:

GMbase
The user created extension library

GMbase is a relatively new site dedicated to hosting the web's largest archive of freely available extensions for Game Maker. Upon learning of this fantastic resource I couldn't resist submitting my most useful extension package - HexScripts. Originally a collection of scripts - as the name implies - this package extends Game Maker's built in binary capabilities to include the reading and writing of big and little endian formatted integer words of any length. It also adds functionality for converting between different bases (such as binary and decimal) and in addition includes a function to move the position of the document relative to the current position.

```
#define file_bin_read_word(fileid,  
size, bigend)
```

file_bin_read_word reads an integer word of the given size (in bytes) from an open file stream in little or big-endian byte order.

```
#define  
file_bin_write_word(fileid, size,  
bigend)
```

file_bin_write_word does exactly what it says, it writes a little or big-endian formatted integer word with the given size (in bytes) to an open binary file.

```
#define radix_change(number,  
old_base, new_base)
```

The radix, or "base" of a number (ex. Decimal, binary, or hexadecimal) is notoriously difficult to change with conventional mathematical methods. Rithiur however, takes a different approach and uses strings to change the base of any given number. I obtained permission long ago to use this script, however, credit to Rithiur should probably be given.

```
#define  
file_bin_seek_relative(fileid,  
position)
```

An extension of the `file_bin_seek` script built into Game Maker, `file_bin_seek_relative` seeks from the current position in a file instead of the absolute position. This means that if you seek for a position of 5 from byte 10, you will move to byte 15 instead of byte 5.

Conclusion

Besides the functions, there are also over 60 constants to help you when defining large bases or data types! The bases follow the "nb_*" naming scheme (eg. `nb_bin`, `nb_dec`, `nb_quin`, `nb_hex` etc.) while the data types follow the "dt_*" convention (eg. `dt_byte`, `dt_long`, `dt_quad`, `dt_short`)

Download the extension

http://gmbase.cubedwater.com/view_ex.php?ex=118

Leif Greenman ■

Tile Optimizer

Tile Optimizer is an amazing script used to optimize tile usage in Game Maker – leading to a speed boost.

The author says the tile optimizer could increase the FPS of a game (provided it uses a sufficient amount of tiles) dramatically – a figure that could rise up to (but not always) an 820% FPS gain.

The script is very simple – however the author warns us not to use this script unless we have about 30 tiles in a room, or more, otherwise the script would not be as effective.

The example is provided in the GM6 format. You must have at least a registered version of Game Maker 6.

Get it now!

<http://xrl.us/tile>

QUICK REVIEW

Script of the Month

SCRIPTS

Powered By:

GMLscripts.com

List Saving and Loading

Though “lists” as a concept – is a great thing for Game Maker developers like me, a problem it has is that it cannot be saved with the regular `game_save()` function.

This means that other methods should be used to save and load lists.

Game Maker has its own set of function that allows you to store a list to a string and later read it. These could be written to files to be saved or loaded.

However if the creator wanted to store lists in a readable and editable form (by humans), the best way to store it would be using this set of scripts:

Save Script

```
/*
** Usage:
**     ds_list_save(dsid,filename,separator)
** Arguments:
**     dsid           ds_list to be saved
**     filename       file path to save the list to
**     separator      string used as separator between
elements (optional)
** Returns:
**     0 if successful, or (-1) on error
** Notes:
**     If separator is not given, each list element will
be on a separate line.
**     If separator also appears within data, the list
will not load correctly.
** GMLscripts.com
*/
{
    var dsid,filename,sep,fid,i;
    dsid = argument0;
    filename = argument1;
    if (is_string(argument2)) sep = argument2; else sep =
chr(13)+chr(10);
    fid = file_text_open_write(filename);
    if (fid > 0) {
        for(i=0; i<ds_list_size(argument0); i+=1) {
            if (i != 0)
file_text_write_string(fid,sep);
file_text_write_string(fid,string(ds_list_find_val
ue(dsid,i)));
        }
    }
```

```
file_text_close(fid);
return 0;
}else{
    return -1;
}
}
```

Load Script

```
/*
** Usage:
**     dsid = ds_list_load(filename,separator);
** Arguments:
**     filename       file path to save the list to
**     separator      string used as separator
between elements (optional)
** Returns:
**     ds_list id if successful, or (-1) on error
** Notes:
**     If separator is not given, each list element
must be on a separate line.
**     If separator also appears within data, the list
will not load correctly.
** GMLscripts.com
*/
{
    var dsid,filename,sep,fid,dat,len,ind,pos;
    filename = argument0;
    if (is_string(argument1)) sep = argument1; else sep
= chr(13)+chr(10);
    fid = file_text_open_read(filename);
    if (fid > 0) {
        dat = "";
        while (!file_text_eof(fid)) {
            dat += file_text_read_string(fid);
            file_text_readln(fid);
        }
        dat += sep;
        len = string_length(sep);
        ind = 0;
        dsid = ds_list_create();
        repeat (string_count(sep,dat)) {
            pos = string_pos(sep,dat)-1;
            ds_list_add(dsid,string_copy(dat,1,pos));
            dat = string_delete(dat,1,pos+len);
            ind += 1;
        }
        file_text_close(fid);
        return dsid;
    }else{
        return -1;
    }
}
```

Contributors

Thanks to **xot** and **Leif902** (Leif Greenman) for creating the script.

Eyas Sharaiha

Graphic Design

"But good gamezzz dont n33d graphix!,,

Most developers using GM forget that their main way of conveying information to the user is through graphics. Nearly all the relevant info the user needs to play the game is given with sprites, backgrounds, text, and all sorts of images. Particles tell you when something is being hit, or maybe when the thrusters are on. Sprites animate to signify that something is moving, or trying to at least. And even when developers know how critical graphics are to their games, they often overlook it. On any given day, you can find a handful of bad games, and a handful of good games. In 95% of the cases, the good games will have clear, easy to see graphics, and it will all look nice and clean. The bad games will have distracting backgrounds, sloppy layouts, and sprites that make your eyes bleed. There is no reason for this! With little knowledge of even MS Paint, anyone can fix up their games to look nice.

Uniformity

All games need to have a standard drawing style for all the graphics. While this can sometimes be difficult with multiple artists, but it's definitely achievable on any project. If anyone takes a look at Company of Heroes, and then takes a look at Command and Conquer, they will definitely notice the different art styles. Use whatever is best for your game! Puzzle games need simple art, that is easy to see and understand. Minesweeper is a great example of this. The game does not have any new next generation blur affects and normal

mapping that makes your computer cringe. However, the game rocks - say what you want – and it rocks without any of those special effects.

Style

Using real photos, or heavily textured graphics in puzzle games can make the game all blend together and too hard to play. They need simplified graphics, that are easily recognized by the player. Some of the best games you will notice, have cartoon-like graphics, with a black 1 pixel outline on everything. This makes it all very easy to see, no matter what the sprites are in front of. Now obviously, an RPG with more detailed graphics will take much more time, but in the end, it will look amazing if done right.

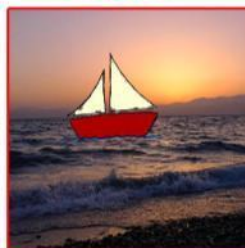
Priority

The colors of your objects is very important in any type of game. Too many game makers use a stunning background that is way too busy for their game. The background is just what it sounds like, it's to be left in the background and not distract from the rest of the game at all.

Effort!

Fixing up graphics really does not take up much time. I hope this article has taught all the developers out there that putting a little more effort into the thought part of their games' graphic design can make all the difference!

BAD



Dan Meinzer ■
GOOD



GRAPHICS



BAD

'Chain Chomp' has a textured dark background. There is no need for a rocky looking background because it has nothing to do with the game and only adds distraction. Even worse, the main character has very few colors, and it very dark like the background. If the game were to use better graphic techniques, it would look more professional.



GOOD

'Seiklus' has cartoon-like sprites that all have a distinct black outline. This makes having a white character on a bright background possible without making it hard to see. The simplicity in the graphics is also consistent throughout the game and nothing looks out of place.



GOOD

Objects in 'Wubly 2' are color coded. The most important objects are lime green, crimson red, or sharp pink, while the walls are left a navy blue, which is closer to the background color.

Sounds in Games: How Far?

EDITORIALS

There is no doubt that sounds and music have a crucial part in games – but the true question is: How much is too much?

I'd certainly like to listen to background music, and I'd certainly like to hear some sound effects whenever I do a certain action, but at what point does sounds become a downgrading experience to the game?

To be able to come to a correct, logical answer – we need to differentiate between various types of sounds – as each one has its own type of limitations.

Sounds include background music, and sound effects. The sound effects group itself could be divided into two: those you listen to when clicking buttons, and interacting with the game's UI, and those you get from the characters themselves: such as the sound of them talking, their steps, gunshots, etc.

Though the article is more aimed at the higher limit for sounds – how much is too much – I'll also be discussing the must-have sounds as well – the 'minimal requirements' for games.

Background Music

Background music is an essential part of a game, and my personal recommendation is: at no point in the game – be it Game Play or just interacting with the interface – should there be no background music at all.

This may be thought of as – there is no "too much" background music, and when it comes to quantity: it's true; but there are certain types of music which



cannot and should not be implemented in certain areas. Though in menus, option screen, and other similar situations, there might be no need for background music, I certainly think there should be.

The type of music to be present at such areas however must – I repeat: **must** – be different (in terms of tone and mode) from what is present in the gameplay. Such music should be calmer, and it must not be loud.

Though it's a good idea to make something "exciting" that would generate some enthusiasm for the player, you must remember it should be low and transparent. That is not the case when the game is being played; sounds are now *part* of the gameplay – and should be louder, more exciting, and be directly related to the tone of the game.

However, even in the Game, you must be very careful that the background music does not degrade the sound effects in the game; you should be able to hear all the players speaking, you should be able to hear the sounds of buttons you're clicking, and steps the players are making.

Sound Effects

Sound effects that need to be in the game form a huge range of different



GM Cash 1.1

GM Cash is a Game Maker 7 extension that adds more Drag n' Drop actions to the object properties window.

The library added allows the game developer to add links and commands that the user could use to donate to the game's developers, or just pay to get a full version of the game.

The extension library supports 8 stores at its current version, these include PayPal, ShareIt, Payson, Ebay, VSTORE, MyStore, Mal's Ecommerce, and OSCommerce. It also has support to be used with the YoYo Games community and the Game Maker Community.

The extension requires Game Maker 7.

Get it now!

<http://xrl.us/gmcash>

QUICK REVIEW

Sounds in Games: How Far?

types of effects. Out of the many possible sounds, I thought the following need to be included in a game:

- Bullets
- Bullets colliding with wall, player, and enemies
- Buttons being pressed
- Steps (preferably for different materials)
- Players taking
- Vehicles

Extra sounds that would be nice to be added – but not required for all games – are those that would add excitement to the game, but without them the game is not “silent”, these include:

- Occasional “Oh, I’m hit!” screams
- Dying sounds
- “Fire in the hole” and similar

screams on various occasions

- “Radio” when in vehicles

Of course, such sound effects do not apply for all kinds of games, but it certainly gives you a picture of what could be included in your games.

One general piece of advice though, sound effects must NOT be overused. If you’re thinking about adding sound effects to something that occurs very frequently, like a button that might be pressed several times in 20 seconds, then you might want to think again.

I know what some might be thinking: what about footsteps? Sure, footsteps are very redundant, and that is where talent kicks in; footsteps should have the same status that I gave to background music in menus: it should be low and transparent – something that could be heard, but does not keep annoying the

player.

Conclusion

Sounds need to be included in the game, without them – a game wouldn’t be as good. The challenge would be knowing what types of sounds you should put in, when they should be put, and most importantly: when to stop.

As I previously said, redundant sounds effects should be avoided, and so should using the same sound effect for multiple actions. As for background music, you should know when it should be low and when it should be higher, when it should be calm and when it should be more exciting, as well as the quality and types of sounds to be used all over.

Eyas Sharaiha ■

FATAL

REVIEWS

FATAL is described as a “fast paced SHMUP”. You’d be hard pressed to argue against that. Within seconds of starting, you find yourself surrounded by bullets from various enemy craft whilst you fire a seemingly inexhaustible armory of bullets. It’s hard to last for long at all and obvious to see why you start with five lives rather than the more traditional three.

You can automatically see that the game will have a retro feel to it. The graphics are formed of very simple, similar looking ships and lines to give a feeling of depth. Similarly, the sound is the collection of squeaks and beeps that one expects to hear off a classic arcade machine. To me, this is a very admirable quality as I’ve always been a fan of very retro game.

The controls are fairly easy to grasp. You

use both hands and are spread out well across the keyboard. The three action keys are close together which may be easy for some, but others may find difficult – slipping and pressing the wrong key or doing so because of not paying enough attention proved problematic for me.

As ever there are cons to the game. Being a work in progress, it is typically lacking in features. But that will undoubtedly change. ChikEn believes that the game will eventually sell for almost 23 USD, so it will almost certainly be abundant in features.

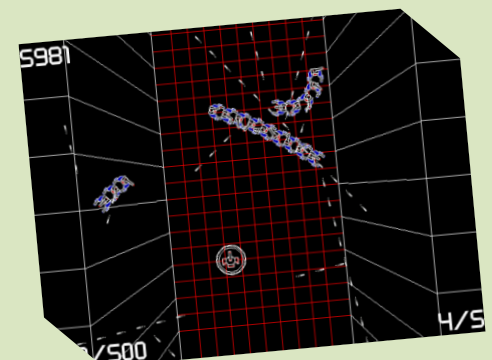
I look forward to the end version of this game. With more craft to choose from and different modes of play, it will be great to play. I probably won’t be buying

the full version, but I hope it will be worth the money. It is certainly one of the most underrated games in the GMC at the moment.

Some Information

Creators: ChikEn AtE mY dOnUtS and Coffee.

Download link: [here](#).



“Grego” Tyler ■

Bounce 2

REVIEWS

What they say

In BOUNCE 2 you control a ball that you have to try and guide to the end of the level. You have to avoid obstacles by bouncing on different colored blocks which alter your bounce height (blue makes you go higher, red lower, etc). You will use many objects and power-ups to reach the end of each level.

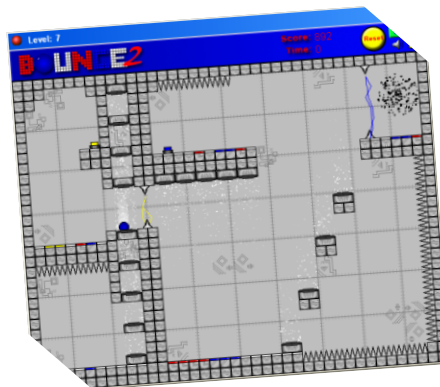
Review

This game takes a simple concept, guiding a ball to the end of a level, and adds in different elements to make the game harder as you progress.

The ball you control is constantly bouncing, by using the left and right arrow keys you can change the direction in which the ball travels. In order to reach the end of a level you have to avoid the many obstacles which lie in your path. At first these are just a maze of walls which the ball must be guided through, but later in the game dangers such as spikes and electricity are added which, if touched, send you back to the start of the level.

height is adjusted enabling you to reach high platforms and duck under obstacles. Blue blocks make the ball bounce higher, whilst red blocks reduce the height. There are also yellow blocks which blast your ball up high so you can reach the portal to the next level.

Some strategic thinking is needed if you are to reach the end of some level successfully on your first attempt.



Blasts of electricity can be turned off by bouncing on the appropriate colored switch and fans can be used to blow your ball both to your advantage and disadvantage.

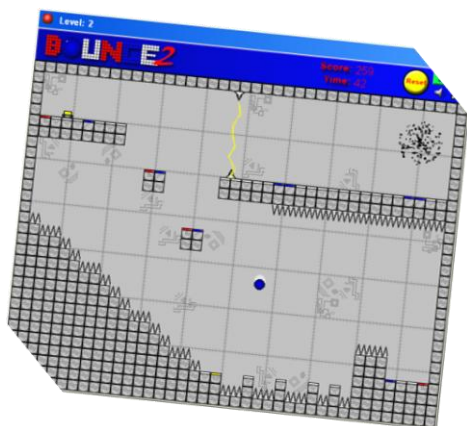
Thankfully a save and load game feature is included, otherwise you may

get bored of playing through the 50 levels (including the levels from the original 'Bounce') consecutively.

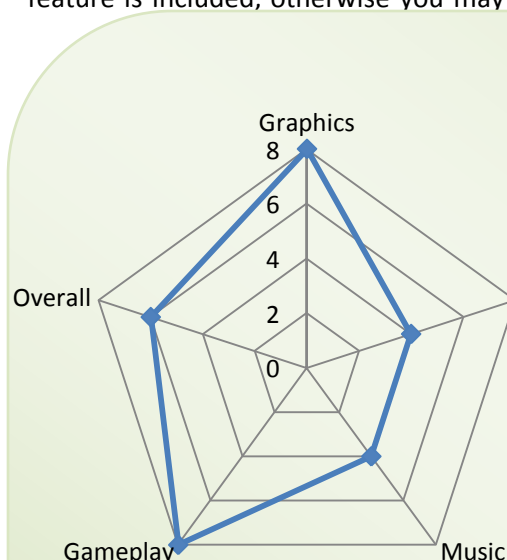
Many who played the game criticized it, saying that if you keep the ball over a red block too long. As the bounce height is constantly reduced eventually the ball becomes stuck and cannot be moved off the block, resulting in an irritating noise being played until the level is reset. Obviously this adds another risk to the game which the player must overcome to complete a level but the opinion of some seems to be that this is too much. Personally I'm not too bothered by it, however I did become stuck a couple of times whilst reviewing this game which was a minor annoyance.

Sounds do become repetitive and you will probably recognize some of them from other applications when you play the game for the first time. Fortunately there are options to toggle both music and sound effects.

Philip Gamble ■



By manoeuvring your ball so it hits different coloured blocks your bounce



Information

Download Size: 2 MB

Download Link: xrl.us/bounce2

Download Type: Zipped Executable

Released: July 2006

Author: SuperCasey4

Review Summary

Graphics: Adequate, not perfect

Sound & Music: Becomes an irritation, too repetitive.

Gameplay: Very simple controls, 'sticky' issue

Overall: A nice idea enhanced by power-ups and obstacles

Snow Ball War

What they say

They say it's a classic game of Capture the Flag. You and a team of blues fight the reds for the flag. Click once to get a snow ball ready, click again to throw. If you get hit you will be sent back to the nearest spawn point. It's primitive but addicting!!

Review

Snow Ball War definitely falls into the category of mini-game. It is based on a concept used in several Flash games where you control a member of a team in the weather dependent sport of snow ball fighting. This has also been seen in games of a similar nature where instead of snow a catapult or grenade can be used to inflict damage on your opposition.

The AI in Snow Ball War is unquestionably clever, it is realistic in the decisions it makes and it isn't easy to avoid your rivals at all. That said by running around obstacles you are able to dodge the snowballs which the red team will through at you, and grab the flag from your opposition's base which makes the game winnable.

Your own team however is not supportive of your efforts to go and grab the enemies' flag, instead tending to throw snowballs at any opposition they see and engage in their own mini war against the computer's team.

Unlike many mini-games there are a number of levels to choose from, 4 in fact, each with its own terrain and a different number of participants.

One snowball can hit multiple people,

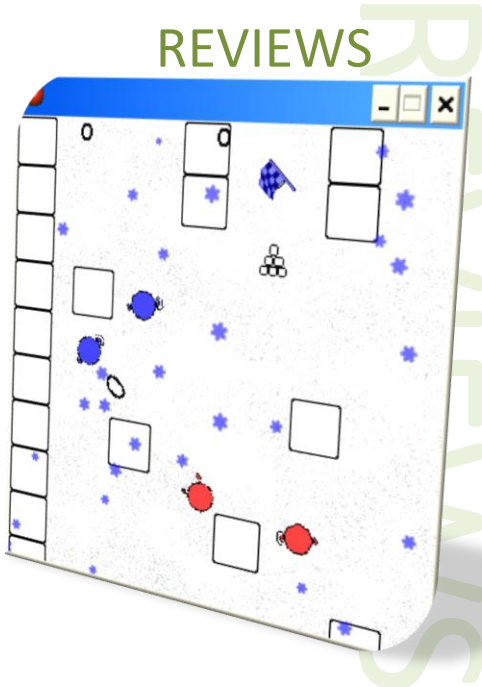
and an added realism is that you can be hit by members of your own team. Perhaps one problem is the one-hit health each snowballer has. You are forced to respawn each time you are snowballed – a health bar indicator above a player with three-hit health would be more realistic and would give you a longer chance of fighting your opposition.

The blocks in the game serve only as

obstacles to navigate around as snowballs can be thrown over them to hit someone on the other side.

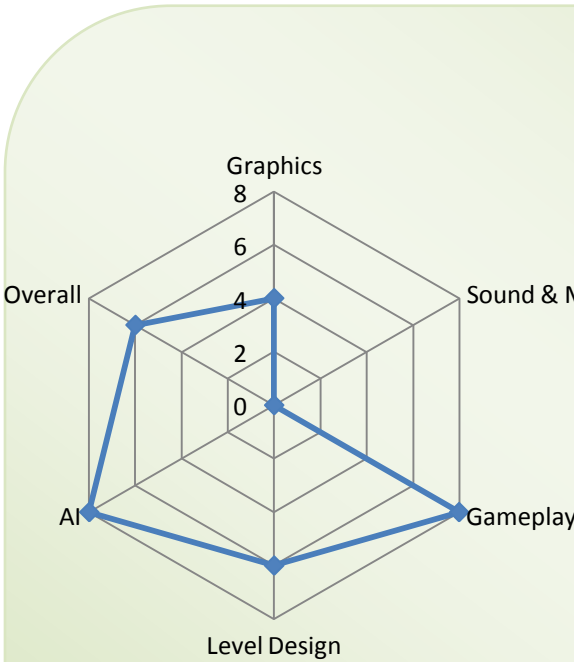
Other realistic features such as the fact that you have to gather snow before you can throw each ball, and that balls can only stay in the air for a certain distance further enhance gameplay.

However it isn't all fun and games, the screen view is very small and whilst the graphics are adequate for basic gameplay they could easily be improved. There is little noticeable



variation in the different levels available either. If more customization was added to this game it could be much more fun, and with such a simple a game a level editor enabling players to create their own snowball arenas would also be a useful addition.

Philip Gamble■



Information

Download Size: 1MB
Download Link: xrl.us/snowball
Download Type: Zipped Executable
Released: July 2007
Author: Beat22

Review Summary

Graphics: Could be easily improved, no much variation
Sound & Music: Non-existent
Gameplay: No real bugs, smooth
AI: Your opposition are effective in both capturing the flag and defending their base. Let down by your own team's choice of tactics.
Overall: Simple concept which should have been built upon further.

SUMMARY

Until Next Time!

And so it ends: another record-breaking, content-packed issue of MarkUp magazine!

MarkUp Issue 6 also celebrates the half-year anniversary of MarkUp magazine! We are thrilled by the amount of support, feedback, and contribution MarkUp has gotten in the past six months.

We are also thrilled about the growth of MarkUp magazine, in terms of content, readership, and yes: community support.

The entire MarkUp staff is overwhelmed with self-satisfaction. After all, we have not only delivered an excellent publication for six months, but also managed to deliver such a content packed issue on-schedule: each time.

every month.

Before I start with all of the thanks, I have an apology first; sorry to Leif Greenman who also was a contributor in the Script of the Month for issue 5 but was not mentioned. Contributors in scripts at GMLscripts.com are now mentioned in articles.

Again I want to say how much I appreciate the support we got from various Game Maker sites. First and foremost, thanks to YoYo Games for the continued endorsement and support of MarkUp.

Thanks to GameMakerBlog.com and GMLscripts.com for the continued contribution to MarkUp Magazine. Also thank you for GameMakerResource.com for all the work that they have done in

THE WRAP UP

the previous issues, and we're looking forward for more participation in the future.

MarkUp has grown in an overwhelming way in the past six months, but we still aspire to more growth which can only be done by your continued support and contributions. To contribute to the magazine you could visit the MarkUp forum [here](#).

The quality of MarkUp magazine is expected to improve continuously as we get more readers and supporters. Remember that even if you cannot contribute in terms of articles to the Magazine, we won't improve if you won't give us your much appreciated opinion.

Once again, thanks for your support!

The MarkUp Staff

Be sure to check out...

GMking.org – the network behind MarkUp – also supports developers in various ways. GMking.org provides Game Maker developers with excellent resources, as well as other developers from other IDEs.

MarkUp has sister projects, also developed and maintained by GMking.org, all meant to help Game Developers. To learn more information about your Game Platform of choice, you could check out GMPedia.org. GMPedia is a game development wiki with a growing community-base and content.

You can also listen to our audcast – GMPod – related to the Game Maker Community and its events by viewing the [Audcast](#) page on the GMking.org main page.

GMking.org

Let them make games!

Markup is an open publication made possible by the contributions of people like you; please visit markup.gmking.org for information on how to contribute. Thank you for your support!

©2007 Markup, a GMking.org project, and its contributors. This work is licensed under the Creative Commons Attribution-Noncommercial-No Derivative Works 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA. Additionally, permission to use figures, tables and brief excerpts from this work in scientific and educational works is hereby granted, provided the source is acknowledged. As well, any use of the material in this work that is determined to be "fair use" under Section 107 or that satisfies the conditions specified in Section 108 of the U.S. Copyright Law (17 USC, as revised by P.L. 94-553) does not require the author's permission.

The names, trademarks, service marks, and logos appearing in this magazine are property of their respective owners, and are not to be used in any advertising or publicity, or otherwise to indicate sponsorship of or affiliation with any product or service. While the information contained in this magazine has been compiled from sources believed to be reliable, GMking.org makes no guarantee as to, and assumes no responsibility for, the correctness, sufficiency, or completeness of such information or recommendations.